THE EXPERT'S VOICE®

SEGUNDA EDICIÓN

Pro Git

TODO LO QUE NECESITAS SABER ACERCA DE GIT

Scott Chacon y Ben Straub

Apress®

Pro Git

Table of Contents

Licence

Prefacio por Scott Chacon

Bienvenidos a la segunda edición de Pro Git. La primera edición fue publicada hace más de cuatro años. Desde entonces mucho ha cambiado aunque muchas cosas importantes no. Mientras que la mayoría de los conceptos y comandos básicos siguen siendo válidos hoy gracias a que el equipo principal de Git es bastante fantástico manteniendo la compatibilidad con versiones anteriores, ha habido algunas adiciones y cambios significativos en la comunidad circundante a Git. La segunda edición de este libro pretende dar respuesta a esos cambios y actualizar el libro para que pueda ser más útil al nuevo usuario.

Cuando escribí la primera edición, Git seguía siendo relativamente difícil de usar y una herramienta escasamente utilizada por algunos hackers. Estaba empezando a cobrar fuerza en algunas comunidades, pero no había alcanzado la ubicuidad que tiene actualmente. Desde entonces, casi todas las comunidades de código abierto lo han adoptado. Git ha hecho un progreso increíble en Windows, en la explosión de interfaces gráficas de usuario para el mismo en todas las plataformas, en soporte IDE y en uso en empresas. El Pro Git de hace cuatro años no trataba nada de eso. Uno de los principales objetivos de esta nueva edición es tocar todas esas nuevas fronteras en la comunidad de Git.

La comunidad de código abierto que usa Git también se ha disparado. Cuando originalmente me puse a escribir el libro hace casi cinco años (me tomó algún tiempo sacar la primera versión), acababa de empezar a trabajar en una empresa muy poco conocida desarrollando un sitio web de alojamiento Git llamada GitHub. En el momento de la publicación había quizás unos pocos miles de personas que utilizaban el sitio y sólo cuatro de nosotros trabajando en él. Al momento de escribir esta introducción, GitHub está anunciando nuestro proyecto alojado número 10 millones, con casi 5 millones de cuentas de desarrollador registradas y más de 230 empleados. Amado u odiado, GitHub ha cambiado en gran medida grandes franjas de la comunidad de código abierto de una manera que era apenas concebible cuando me senté a escribir la primera edición.

Escribí una pequeña sección en la versión original de Pro Git sobre GitHub como un ejemplo de Git hospedado con la cual nunca me sentí muy cómodo. No me gustaba estar escribiendo sobre lo que esencialmente consideraba un recurso comunitario y también hablando de mi empresa. Aunque aún me desagrada ese conflicto de intereses, la importancia de GitHub en la comunidad Git es inevitable. En lugar de un ejemplo de alojamiento Git, he decidido desarrollar esa parte del libro más detalladamente describiendo lo qué GitHub es y cómo utilizarlo de forma eficaz. Si vas a aprender a usar Git entonces saber cómo utilizar GitHub te ayudará a tomar parte en una comunidad enorme, que es valiosa no importa qué alojamiento Git decidas utilizar para tu propio código.

El otro gran cambio en el tiempo transcurrido desde la última publicación ha sido el desarrollo y aumento del protocolo HTTP para las transacciones de red de Git. La mayoría de los ejemplos en el libro han sido cambiados a HTTP desde SSH porque es mucho más sencillo.

Ha sido increíble ver a Git crecer en los últimos años a partir de un sistema de control de versiones relativamente desconocido a uno que domina básicamente el control de versiones comerciales y de código abierto. Estoy feliz de que Pro Git lo haya hecho tan bien y también haya sido capaz de ser uno de los pocos libros técnicos en el mercado que es a la vez bastante exitoso y completamente de código abierto.

Espero que disfruten de esta edición actualizada de Pro Git.

Prefacio por Ben Straub

La primera edición de este libro es lo que me enganchó a Git. Ésta fue mi introducción a un estilo de hacer software que se sentía más natural que todo lo que había visto antes. Había sido desarrollador durante varios años para entonces, pero éste fue el giro que me envió por un camino mucho más interesante que el que había seguido.

Ahora, años después, soy contribuyente a una de las principales implementaciones de Git, he trabajado para la empresa más grande de alojamiento Git, y he viajado por el mundo enseñando a la gente acerca de Git. Cuando Scott me preguntó si estaría interesado en trabajar en la segunda edición, ni siquiera me lo pensé.

Ha sido un gran placer y un privilegio trabajar en este libro. Espero que le ayude tanto como lo hizo conmigo.

Dedicatorias

A mi esposa, Becky, sin la cual esta aventura nunca hubiera comenzado. - Ben

Esta edición está dedicada a mis niñas. A mi esposa Jessica que me ha apoyado durante todos estos años y a mi hija Josefina, que me apoyará cuando esté demasiado mayor para saber lo que pasa. - Scott

Contribuidores

Debido a que este es un libro cuya traducción es "Open Source", hemos recibido la colaboración de muchas personas a lo largo de los últimos años. A continuación hay una lista de todas las personas que han contribuido en la traducción del libro al idioma español. Muchas gracias a todos por colaborar a mejorar este libro para el beneficio de todos los hispanohablantes.

```
amabelster                      Fernando Guerra                Mario Rincon
andres-mancera                  German Gonzalez                michaelizer
Andres Mancera                  Gytree                         Moises Ariel Hernández
Rojo
blasillo                        herrmartell                    moisesroj0
Carlos A. Henri  quez Q          Jean-Noël Avila                pilarArr
Christopher Díaz                José Antonio Muñoz Jiménez     Pilar Arr
Christopher Díaz Riveros        José Carlos García            Samuel Castillo
David Bucci                     josue33                        Santiago Aramis
David Munoz                     Juane99                        Vladimir Rodríguez
devwebcl                        Juanjo Amor                    Xxdaniels751xX
Eliecer Daza                    Juan Miguel Jimenez            yesmin41
Enrique Matías Sánchez (Quique) Juan Sebastián Casallas        YoandyShyno
```

Introducción

Estás a punto de pasar varias horas de tu vida leyendo acerca de Git. Dediquemos un minuto a explicar lo que tenemos preparado para ti. Éste es un breve resumen de los diez capítulos y tres apéndices de este libro.

En el **Capítulo 1**, cubriremos los Sistemas de Control de Versiones (VCSs, en sus siglas en inglés) y los fundamentos de Git -ninguna cosa técnica, sólo lo que es Git, por qué tuvo lugar en una tierra llena de VCSs, que lo diferencia, y por qué tantas personas lo están utilizando. A continuación, explicaremos cómo descargar Git y configurarlo para el primer uso si no lo tienes ya en tu sistema.

En el **Capítulo 2**, repasaremos el uso básico de Git -cómo usar Git en el 80% de los casos que encontrarás con más frecuencia. Después de leer este capítulo, deberías ser capaz de clonar un repositorio, ver lo que ha ocurrido en la historia del proyecto, modificar archivos, y contribuir cambios. Si el libro arde espontáneamente en este punto, ya deberías estar lo suficientemente ducho en el uso de Git mientras buscas otra copia.

El **Capítulo 3** trata sobre el modelo de ramificación (branching) en Git, a menudo descrito como la característica asesina de Git. Aquí aprenderás lo que realmente diferencia Git del resto. Cuando hayas terminado, puedes sentir la necesidad de pasar un momento tranquilo ponderando cómo has vivido antes de que la ramificación de Git formará parte de tu vida.

El **Capítulo 4** cubrirá Git en el servidor. Este capítulo es para aquellos que deseen configurar Git dentro de su organización o en su propio servidor personal para la colaboración. También exploraremos diversas opciones hospedadas por si prefieres dejar que otra persona lo gestione por ti.

El **Capítulo 5** repasará con todo detalle diversos flujos de trabajo distribuidos y cómo llevarlos a cabo con Git. Cuando hayas terminado con este capítulo, deberías ser capaz de trabajar como un experto con múltiples repositorios remotos, usar Git a través de correo electrónico y manejar hábilmente numerosas ramas remotas y parches aportados.

El **Capítulo 6** cubre el servicio de alojamiento GitHub e interfaz en profundidad. Cubrimos el registro y gestión de una cuenta, creación y uso de repositorios Git, flujos de trabajo comunes para contribuir a proyectos y aceptar contribuciones a los tuyos, la interfaz de GitHub y un montón de pequeños consejos para hacer tu vida más fácil en general.

El **Capítulo 7** es sobre comandos avanzados de Git. Aquí aprenderás acerca de temas como el dominio del temido comando *reset*, el uso de la búsqueda binaria para identificar errores, la edición de la historia, la selección de revisión en detalle, y mucho más. Este capítulo completará tu conocimiento de Git para que puedas ser verdaderamente un maestro.

El **Capítulo 8** es sobre la configuración de tu entorno Git personalizado. Esto incluye la creación de *hook scripts* para hacer cumplir o alentar políticas personalizadas y el uso de valores de configuración de entorno para que puedas trabajar de la forma que deseas. También cubriremos la construcción de tu propio conjunto de scripts para hacer cumplir una política personalizada.

El **Capítulo 9** trata de Git y otros VCSs. Esto incluye el uso de Git en un mundo de Subversion (SVN) y la conversión de proyectos de otros VCSs a Git. Una gran cantidad de organizaciones siguen

utilizando SVN y no van a cambiar, pero en este punto aprenderás el increíble poder de Git, y este capítulo te muestra cómo hacer frente si todavía tienes que utilizar un servidor SVN. También cubrimos cómo importar proyectos desde varios sistemas diferentes en caso de que convenzas a todo el mundo para dar el salto.

El **Capítulo 10** se adentra en las oscuras aunque hermosas profundidades del interior de Git. Ahora que sabes todo sobre Git y puedes manejarte con él con poder y gracia, puedes pasar a estudiar cómo Git almacena sus objetos, qué es el modelo de objetos, detalles de *packfiles*, protocolos de servidor, y mucho más. A lo largo del libro, nos referiremos a las secciones de este capítulo por si te apetece profundizar en ese punto; pero si eres como nosotros y quieres sumergirse en los detalles técnicos, es posible que desees leer el Capítulo 10 en primer lugar. Lo dejamos a tu elección.

En el **Apéndice A** nos fijamos en una serie de ejemplos de uso de Git en diversos entornos específicos. Cubrimos un número de diferentes interfaces gráficas de usuario y entornos de programación IDE en los que es posible que desees usar Git y lo que está disponible para ti. Si estas interesado en una visión general del uso de Git en tu shell, en Visual Studio o Eclipse, echa un vistazo aquí.

En el **Apéndice B** exploramos la extensión y scripting de Git a través de herramientas como libgit2 y JGit. Si estás interesado en escribir herramientas personalizadas complejas y rápidas y necesitas acceso a bajo nivel de Git, aquí es donde puedes ver una panorámica.

Finalmente, en el **Apéndice C** repasaremos todos los comandos importantes de Git uno a uno y reseñaremos el lugar en el libro donde fueron tratados y lo que hicimos con ellos. Si quieres saber en qué parte del libro se utilizó algún comando específico de Git puedes buscarlo aquí.

Empecemos.

Inicio - Sobre el Control de Versiones

Este capítulo se va a hablar de cómo comenzar a utilizar Git. Empezaremos describiendo algunos conceptos básicos sobre las herramientas de control de versiones; después, trataremos de explicar cómo hacer que Git funcione en tu sistema; finalmente, exploraremos cómo configurarlo para empezar a trabajar con él. Al final de este capítulo deberás entender las razones por las cuales Git existe y conviene que lo uses, y deberás tener todo preparado para comenzar.

Acerca del Control de Versiones

¿Qué es un control de versiones, y por qué debería importarte? Un control de versiones es un sistema que registra los cambios realizados en un archivo o conjunto de archivos a lo largo del tiempo, de modo que puedas recuperar versiones específicas más adelante. Aunque en los ejemplos de este libro usarás archivos de código fuente como aquellos cuya versión está siendo controlada, en realidad puedes hacer lo mismo con casi cualquier tipo de archivo que encuentres en una computadora.

Si eres diseñador gráfico o de web y quieres mantener cada versión de una imagen o diseño (es algo que sin duda vas a querer), usar un sistema de control de versiones (VCS por sus siglas en inglés) es una muy decisión muy acertada. Dicho sistema te permite regresar a versiones anteriores de tus archivos, regresar a una versión anterior del proyecto completo, comparar cambios a lo largo del tiempo, ver quién modificó por última vez algo que pueda estar causando problemas, ver quién introdujo un problema y cuándo, y mucho más. Usar un VCS también significa generalmente que si arruinas o pierdes archivos, será posible recuperarlos fácilmente. Adicionalmente, obtendrás todos estos beneficios a un costo muy bajo.

Sistemas de Control de Versiones Locales

Un método de control de versiones usado por muchas personas es copiar los archivos a otro directorio (quizás indicando la fecha y hora en que lo hicieron, si son ingeniosos). Este método es muy común porque es muy sencillo, pero también es tremendamente propenso a errores. Es fácil olvidar en qué directorio te encuentras, y guardar accidentalmente en el archivo equivocado o sobrescribir archivos que no querías.

Para afrontar este problema los programadores desarrollaron hace tiempo VCS locales que contenían una simple base de datos en la que se llevaba el registro de todos los cambios realizados a los archivos.

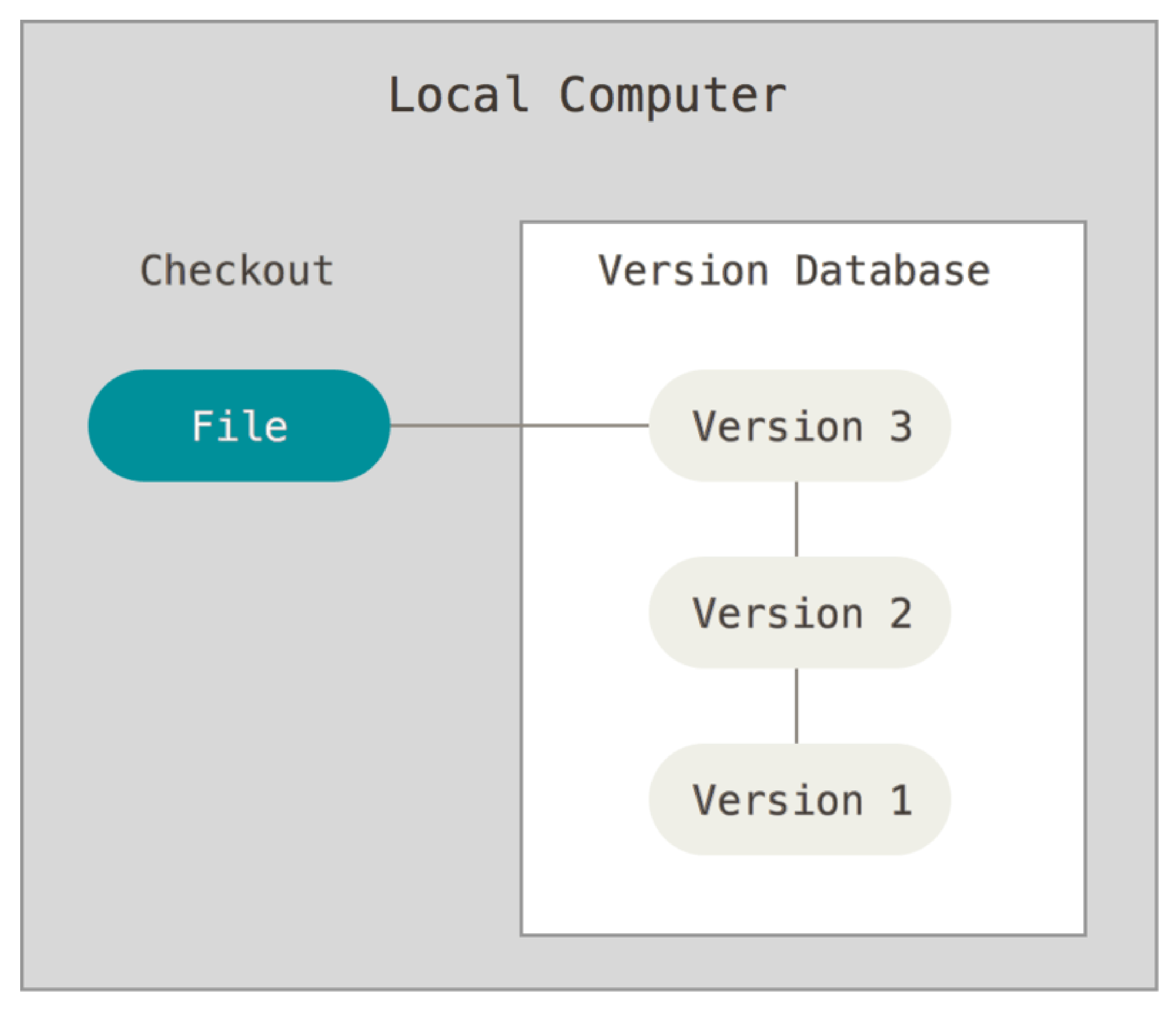


Figure 1. control de versiones local.

Una de las herramientas de control de versiones más popular fue un sistema llamado RCS, que todavía podemos encontrar en muchas de las computadoras actuales. Incluso el famoso sistema operativo Mac OS X incluye el comando `rcs` cuando instalas las herramientas de desarrollo. Esta herramienta funciona guardando conjuntos de parches (es decir, las diferencias entre archivos) en un formato especial en disco, y es capaz de recrear cómo era un archivo en cualquier momento a partir de dichos parches.

Sistemas de Control de Versiones Centralizados

El siguiente gran problema con el que se encuentran las personas es que necesitan colaborar con desarrolladores en otros sistemas. Los sistemas de Control de Versiones Centralizados (CVCS por sus siglas en inglés) fueron desarrollados para solucionar este problema. Estos sistemas, como CVS, Subversion, y Perforce, tienen un único servidor que contiene todos los archivos versionados, y varios clientes que descargan los archivos desde ese lugar central. Este ha sido el estándar para el control de versiones por muchos años.

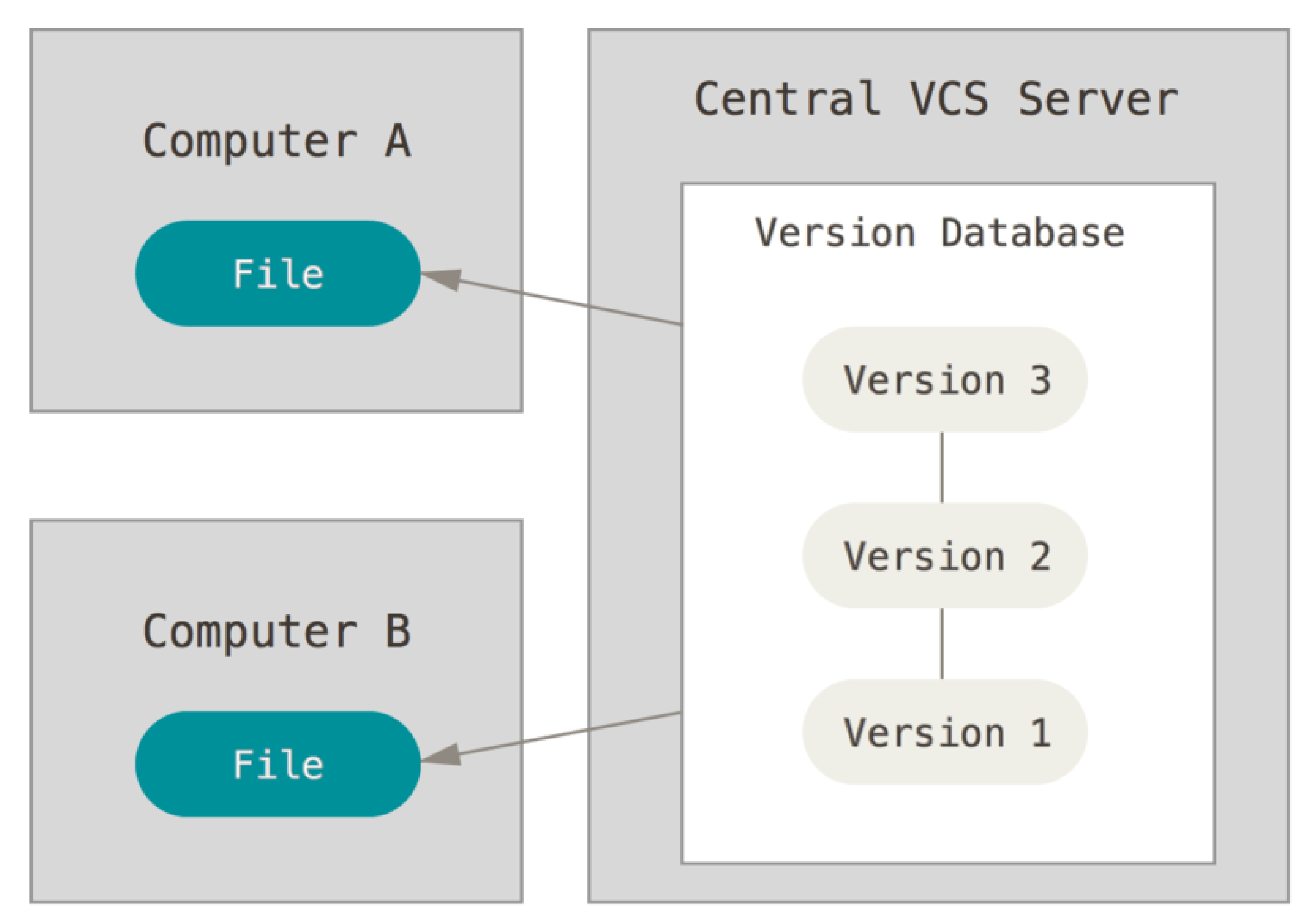


Figure 2. Control de versiones centralizado.

Esta configuración ofrece muchas ventajas, especialmente frente a VCS locales. Por ejemplo, todas las personas saben hasta cierto punto en qué están trabajando los otros colaboradores del proyecto. Los administradores tienen control detallado sobre qué puede hacer cada usuario, y es mucho más fácil administrar un CVCS que tener que lidiar con bases de datos locales en cada cliente.

Sin embargo, esta configuración también tiene serias desventajas. La más obvia es el punto único de fallo que representa el servidor centralizado. Si ese servidor se cae durante una hora, entonces durante esa hora nadie podrá colaborar o guardar cambios en archivos en los que hayan estado trabajando. Si el disco duro en el que se encuentra la base de datos central se corrompe, y no se han realizado copias de seguridad adecuadamente, se perderá toda la información del proyecto, con excepción de las copias instantáneas que las personas tengan en sus máquinas locales. Los VCS locales sufren de este mismo problema: Cuando tienes toda la historia del proyecto en un mismo lugar, te arriesgas a perderlo todo.

Sistemas de Control de Versiones Distribuidos

Los sistemas de Control de Versiones Distribuidos (DVCS por sus siglas en inglés) ofrecen soluciones para los problemas que han sido mencionados. En un DVCS (como Git, Mercurial, Bazaar o Darcs), los clientes no solo descargan la última copia instantánea de los archivos, sino que se replica completamente el repositorio. De esta manera, si un servidor deja de funcionar y estos sistemas estaban colaborando a través de él, cualquiera de los repositorios disponibles en los clientes puede ser copiado al servidor con el fin de restaurarlo. Cada clon es realmente una copia completa de todos los datos.

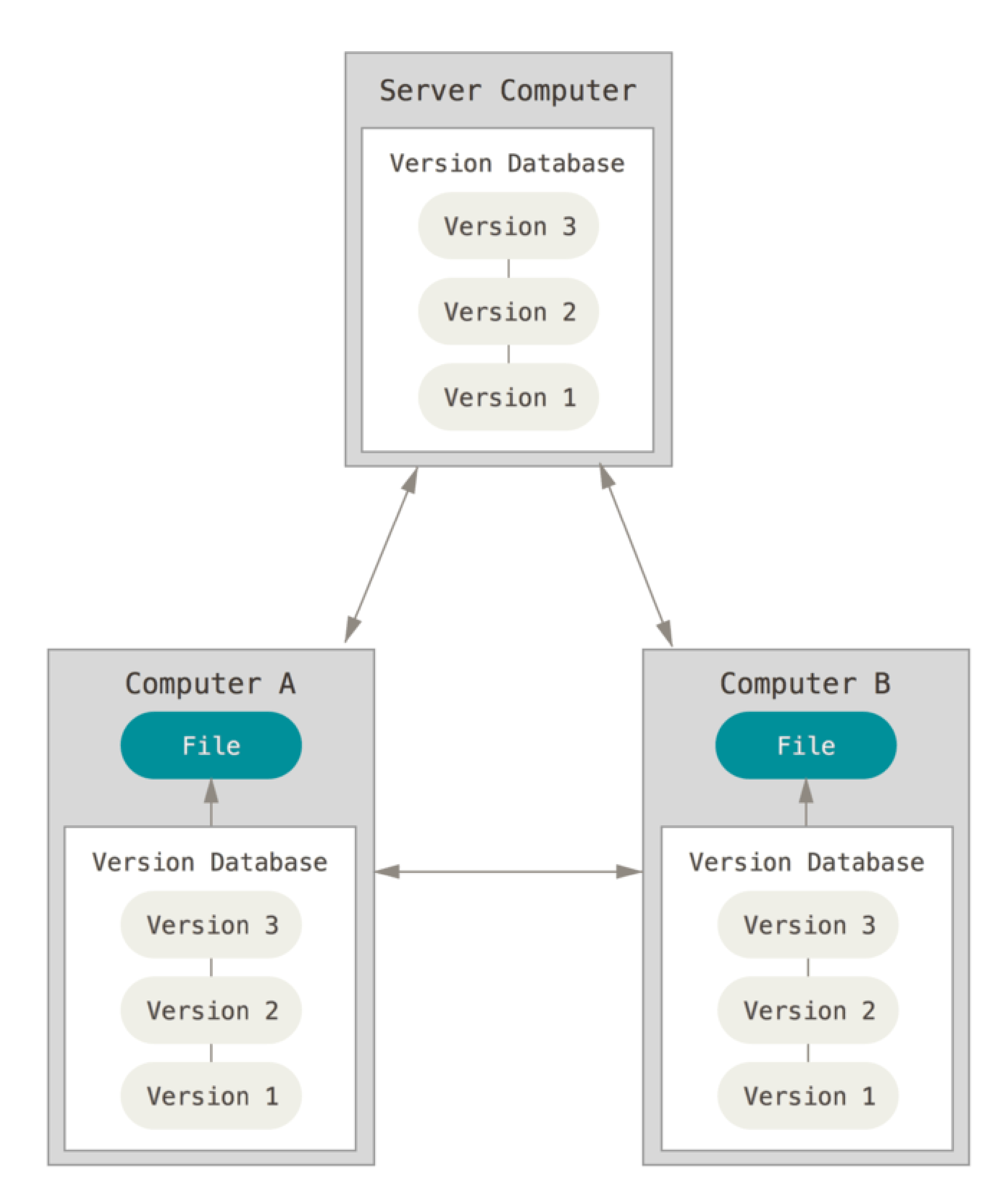


Figure 3. Control de versiones distribuido.

Además, muchos de estos sistemas se encargan de manejar numerosos repositorios remotos con los cuales pueden trabajar, de tal forma que puedes colaborar simultáneamente con diferentes grupos de personas en distintas maneras dentro del mismo proyecto. Esto permite establecer varios flujos de trabajo que no son posibles en sistemas centralizados, como pueden ser los modelos jerárquicos.

Una breve historia de Git

Como muchas de las grandes cosas en esta vida, Git comenzó con un poco de destrucción creativa y una gran polémica.

El kernel de Linux es un proyecto de software de código abierto con un alcance bastante amplio. Durante la mayor parte del mantenimiento del kernel de Linux (1991-2002), los cambios en el software se realizaban a través de parches y archivos. En el 2002, el proyecto del kernel de Linux empezó a usar un DVCS propietario llamado BitKeeper.

En el 2005, la relación entre la comunidad que desarrollaba el kernel de Linux y la compañía que desarrollaba BitKeeper se vino abajo, y la herramienta dejó de ser ofrecida de manera gratuita. Esto impulsó a la comunidad de desarrollo de Linux (y en particular a Linus Torvalds, el creador de Linux) a desarrollar su propia herramienta basada en algunas de las lecciones que aprendieron mientras usaban BitKeeper. Algunos de los objetivos del nuevo sistema fueron los siguientes:

- Velocidad
- Diseño sencillo
- Gran soporte para desarrollo no lineal (miles de ramas paralelas)
- Completamente distribuido
- Capaz de manejar grandes proyectos (como el kernel de Linux) eficientemente (velocidad y tamaño de los datos)

Desde su nacimiento en el 2005, Git ha evolucionado y madurado para ser fácil de usar y conservar sus características iniciales. Es tremendamente rápido, muy eficiente con grandes proyectos, y tiene un increíble sistema de ramificación (branching) para desarrollo no lineal (véase Ramificaciones en Git)

Fundamentos de Git

Entonces, ¿qué es Git en pocas palabras? Es muy importante entender bien esta sección, porque si entiendes lo que es Git y los fundamentos de cómo funciona, probablemente te será mucho más fácil usar Git efectivamente. A medida que aprendas Git, intenta olvidar todo lo que posiblemente conoces acerca de otros VCS como Subversion y Perforce. Hacer esto te ayudará a evitar confusiones sutiles a la hora de utilizar la herramienta. Git almacena y maneja la información de forma muy diferente a esos otros sistemas, a pesar de que su interfaz de usuario es bastante similar. Comprender esas diferencias evitará que te confundas a la hora de usarlo.

Copias instantáneas, no diferencias

La principal diferencia entre Git y cualquier otro VCS (incluyendo Subversion y sus amigos) es la forma en la que manejan sus datos. Conceptualmente, la mayoría de los otros sistemas almacenan la información como una lista de cambios en los archivos. Estos sistemas (CVS, Subversion, Perforce, Bazaar, etc.) manejan la información que almacenan como un conjunto de archivos y las modificaciones hechas a cada uno de ellos a través del tiempo.



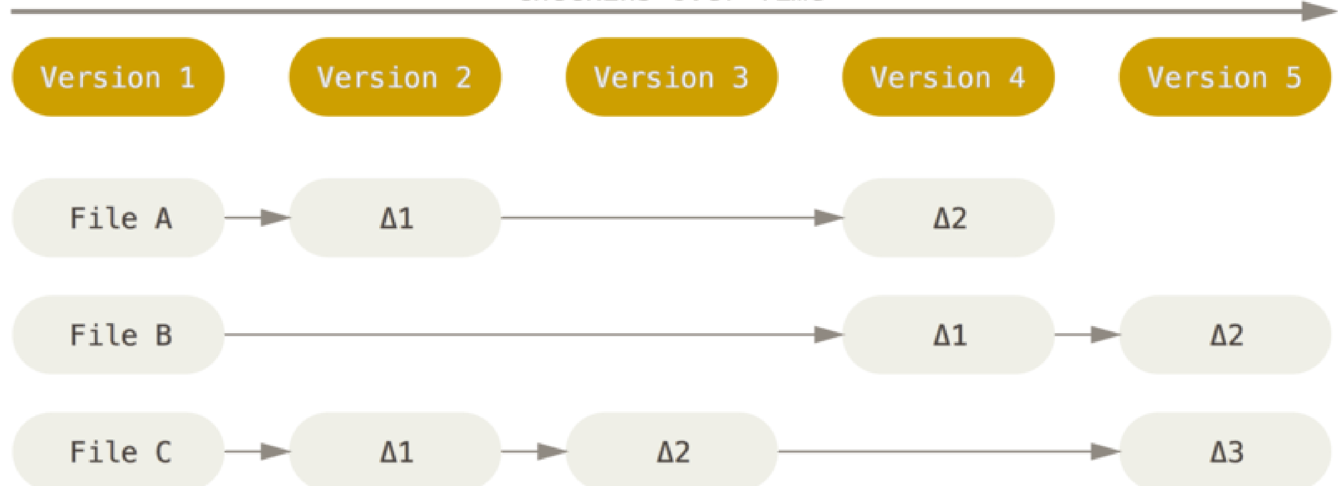


Figure 4. Almacenamiento de datos como cambios en una versión de la base de cada archivo.

Git no maneja ni almacena sus datos de esta forma. Git maneja sus datos como un conjunto de copias instantáneas de un sistema de archivos miniatura. Cada vez que confirmas un cambio, o guardas el estado de tu proyecto en Git, él básicamente toma una foto del aspecto de todos tus archivos en ese momento, y guarda una referencia a esa copia instantánea. Para ser eficiente, si los archivos no se han modificado Git no almacena el archivo de nuevo, sino un enlace al archivo anterior idéntico que ya tiene almacenado. Git maneja sus datos como una secuencia de copias instantáneas.

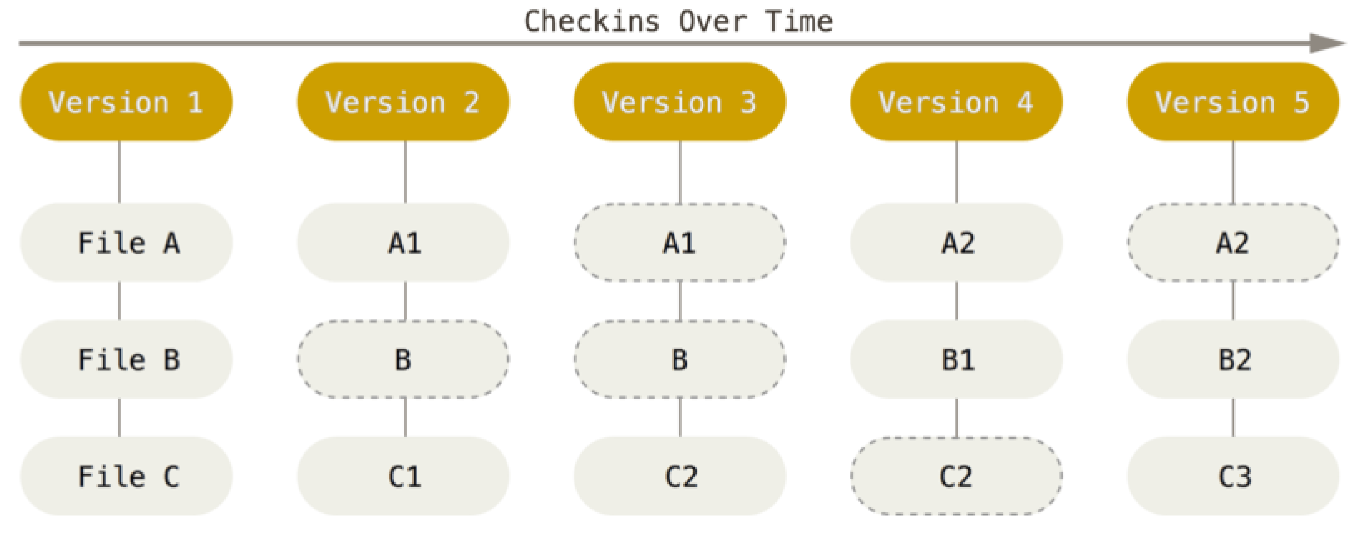


Figure 5. Almacenamiento de datos como instantáneas del proyecto a través del tiempo.

Esta es una diferencia importante entre Git y prácticamente todos los demás VCS. Hace que Git reconsidere casi todos los aspectos del control de versiones que muchos de los demás sistemas copiaron de la generación anterior. Esto hace que Git se parezca más a un sistema de archivos miniatura con algunas herramientas tremendamente poderosas desarrolladas sobre él, que a un VCS. Exploraremos algunos de los beneficios que obtienes al modelar tus datos de esta manera cuando veamos ramificación (branching) en Git en el (véase Ramificaciones en Git).

Casi todas las operaciones son locales

La mayoría de las operaciones en Git sólo necesitan archivos y recursos locales para funcionar. Por lo general no se necesita información de ningún otro computador de tu red. Si estás acostumbrado a un CVCS donde la mayoría de las operaciones tienen el costo adicional del retardo de la red, este

aspecto de Git te va a hacer pensar que los dioses de la velocidad han bendecido Git con poderes sobrenaturales. Debido a que tienes toda la historia del proyecto ahí mismo, en tu disco local, la mayoría de las operaciones parecen prácticamente inmediatas.

Por ejemplo, para navegar por la historia del proyecto, Git no necesita conectarse al servidor para obtener la historia y mostrártela - simplemente la lee directamente de tu base de datos local. Esto significa que ves la historia del proyecto casi instantáneamente. Si quieres ver los cambios introducidos en un archivo entre la versión actual y la de hace un mes, Git puede buscar el archivo hace un mes y hacer un cálculo de diferencias localmente, en lugar de tener que pedirle a un servidor remoto que lo haga u obtener una versión antigua desde la red y hacerlo de manera local.

Esto también significa que hay muy poco que no puedes hacer si estás desconectado o sin VPN. Si te subes a un avión o a un tren y quieres trabajar un poco, puedes confirmar tus cambios felizmente hasta que consigas una conexión de red para subirlos. Si te vas a casa y no consigues que tu cliente VPN funcione correctamente, puedes seguir trabajando. En muchos otros sistemas, esto es imposible o muy engorroso. En Perforce, por ejemplo, no puedes hacer mucho cuando no estás conectado al servidor. En Subversion y CVS, puedes editar archivos, pero no puedes confirmar los cambios a tu base de datos (porque tu base de datos no tiene conexión). Esto puede no parecer gran cosa, pero te sorprendería la diferencia que puede suponer.

Git tiene integridad

Todo en Git es verificado mediante una suma de comprobación (checksum en inglés) antes de ser almacenado, y es identificado a partir de ese momento mediante dicha suma. Esto significa que es imposible cambiar los contenidos de cualquier archivo o directorio sin que Git lo sepa. Esta funcionalidad está integrada en Git al más bajo nivel y es parte integral de su filosofía. No puedes perder información durante su transmisión o sufrir corrupción de archivos sin que Git sea capaz de detectarlo.

El mecanismo que usa Git para generar esta suma de comprobación se conoce como hash SHA-1. Se trata de una cadena de 40 caracteres hexadecimales (0-9 y a-f), y se calcula en base a los contenidos del archivo o estructura del directorio en Git. Un hash SHA-1 se ve de la siguiente forma:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Verás estos valores hash por todos lados en Git porque son usados con mucha frecuencia. De hecho, Git guarda todo no por nombre de archivo, sino por el valor hash de sus contenidos.

Git generalmente solo añade información

Cuando realizas acciones en Git, casi todas ellas solo añaden información a la base de datos de Git. Es muy difícil conseguir que el sistema haga algo que no se pueda enmendar, o que de algún modo borre información. Como en cualquier VCS, puedes perder o estropear cambios que no has confirmado todavía. Pero después de confirmar una copia instantánea en Git es muy difícil de perderla, especialmente si envías tu base de datos a otro repositorio con regularidad.

Esto hace que usar Git sea un placer, porque sabemos que podemos experimentar sin peligro de estropear gravemente las cosas. Para un análisis más exhaustivo de cómo almacena Git su

información y cómo puedes recuperar datos aparentemente perdidos, ver Deshacer Cosas.

Los Tres Estados

Ahora presta atención. Esto es lo más importante que debes recordar acerca de Git si quieres que el resto de tu proceso de aprendizaje prosiga sin problemas. Git tiene tres estados principales en los que se pueden encontrar tus archivos: confirmado (committed), modificado (modified), y preparado (staged). Confirmado significa que los datos están almacenados de manera segura en tu base de datos local. Modificado significa que has modificado el archivo pero todavía no lo has confirmado a tu base de datos. Preparado significa que has marcado un archivo modificado en su versión actual para que vaya en tu próxima confirmación.

Esto nos lleva a las tres secciones principales de un proyecto de Git: El directorio de Git (Git directory), el directorio de trabajo (working directory), y el área de preparación (staging area).

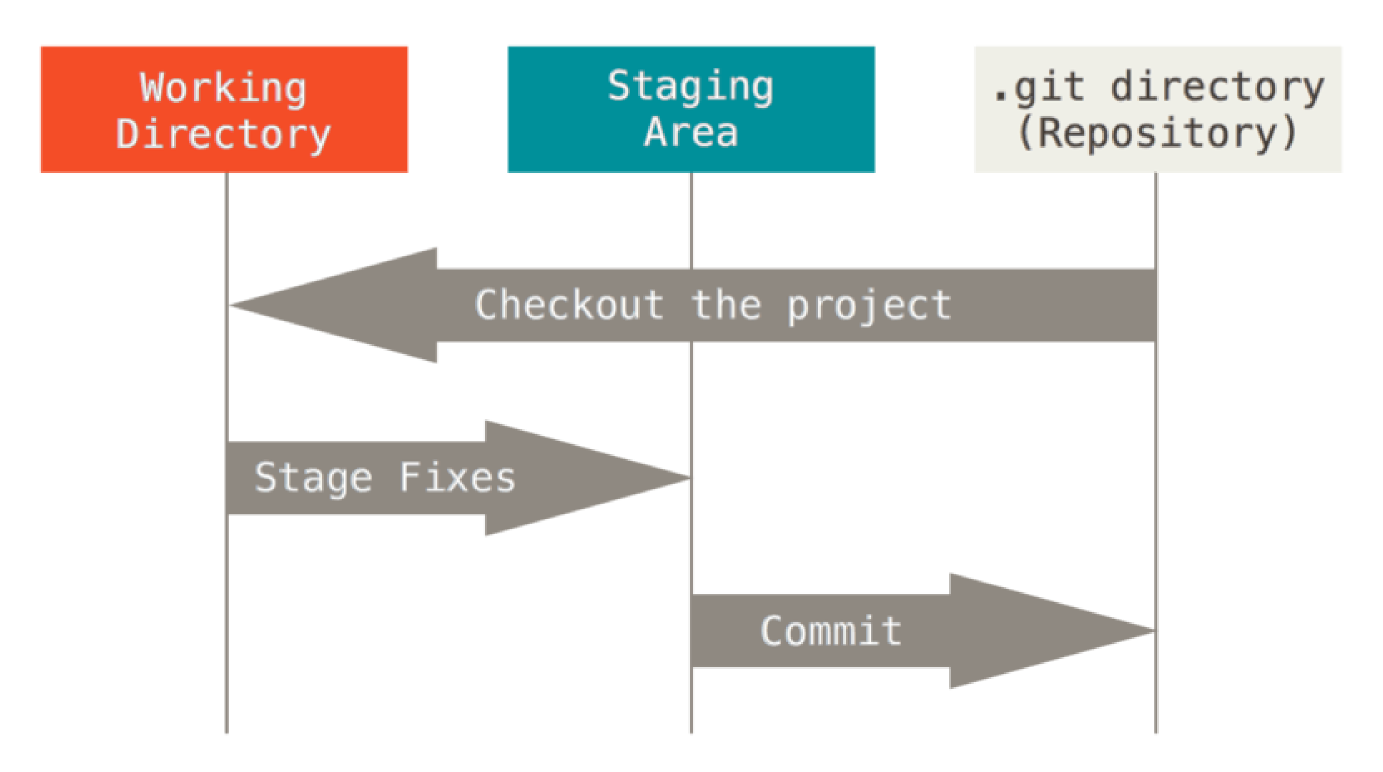


Figure 6. Directorio de trabajo, área de almacenamiento, y el directorio Git.

El directorio de Git es donde se almacenan los metadatos y la base de datos de objetos para tu proyecto. Es la parte más importante de Git, y es lo que se copia cuando clonas un repositorio desde otra computadora.

El directorio de trabajo es una copia de una versión del proyecto. Estos archivos se sacan de la base de datos comprimida en el directorio de Git, y se colocan en disco para que los puedas usar o modificar.

El área de preparación es un archivo, generalmente contenido en tu directorio de Git, que almacena información acerca de lo que va a ir en tu próxima confirmación. A veces se le denomina índice ("index"), pero se está convirtiendo en estándar el referirse a ella como el área de preparación.

El flujo de trabajo básico en Git es algo así:

1. Modificas una serie de archivos en tu directorio de trabajo.

2. Preparas los archivos, añadiéndolos a tu área de preparación.
3. Confirmas los cambios, lo que toma los archivos tal y como están en el área de preparación y almacena esa copia instantánea de manera permanente en tu directorio de Git.

Si una versión concreta de un archivo está en el directorio de Git, se considera confirmada (committed). Si ha sufrido cambios desde que se obtuvo del repositorio, pero ha sido añadida al área de preparación, está preparada (staged). Y si ha sufrido cambios desde que se obtuvo del repositorio, pero no se ha preparado, está modificada (modified). En Fundamentos de Git aprenderás más acerca de estos estados y de cómo puedes aprovecharlos o saltarte toda la parte de preparación.

La Línea de Comandos

Existen muchas formas de usar Git. Por un lado tenemos las herramientas originales de línea de comandos, y por otro lado tenemos una gran variedad de interfaces de usuario con distintas capacidades. En ese libro vamos a utilizar Git desde la línea de comandos. La línea de comandos en el único lugar en donde puedes ejecutar **todos** los comandos de Git - la mayoría de interfaces gráficas de usuario solo implementan una parte de las características de Git por motivos de simplicidad. Si tú sabes cómo realizar algo desde la línea de comandos, seguramente serás capaz de averiguar cómo hacer lo mismo desde una interfaz gráfica. Sin embargo, la relación opuesta no es necesariamente cierta. Así mismo, la decisión de qué cliente gráfico utilizar depende totalmente de tu gusto, pero *todos* los usuarios tendrán las herramientas de línea de comandos instaladas y disponibles.

Nosotros esperamos que sepas cómo abrir el Terminal en Mac, o el "Command Prompt" o "Powershell" en Windows. Si no entiendes de lo que estamos hablando aquí, te recomendamos que hagas una pausa para investigar acerca de esto de tal forma que puedas entender el resto de las explicaciones y descripciones que siguen en este libro.

Instalación de Git

Antes de empezar a utilizar Git, tienes que instalarlo en tu computadora. Incluso si ya está instalado, este es posiblemente un buen momento para actualizarlo a su última versión. Puedes instalarlo como un paquete, a partir de un archivo instalador, o bajando el código fuente y compilándolo tú mismo.

NOTE	Este libro fue escrito utilizando la versión **2.0.0** de Git. Aun cuando la mayoría de comandos que usaremos deben funcionar en versiones más antiguas de Git, es posible que algunos de ellos no funcionen o funcionen ligeramente diferente si estás utilizando una versión anterior de Git. Debido a que Git es particularmente bueno en preservar compatibilidad hacia atrás, cualquier versión posterior a 2.0 debe funcionar bien.

Instalación en Linux

Si quieres instalar Git en Linux a través de un instalador binario, en general puedes hacerlo mediante la herramienta básica de administración de paquetes que trae tu distribución. Si estás en

Fedora por ejemplo, puedes usar yum:

```
$ yum install git
```

Si estás en una distribución basada en Debian como Ubuntu, puedes usar apt-get:

```
$ apt-get install git
```

Para opciones adicionales, la página web de Git tiene instrucciones para la instalación en diferentes tipos de Unix. Puedes encontrar esta información en http://git-scm.com/download/linux.

Instalación en Mac

Hay varias maneras de instalar Git en un Mac. Probablemente la más sencilla es instalando las herramientas Xcode de Línea de Comandos. En Mavericks (10.9) o superior puedes hacer esto desde el Terminal si intentas ejecutar *git* por primera vez. Si no lo tienes instalado, te preguntará si deseas instalarlo.

Si deseas una versión más actualizada, puedes hacerlo partir de un instalador binario. Un instalador de Git para OSX es mantenido en la página web de Git. Lo puedes descargar en http://git-scm.com/download/mac.

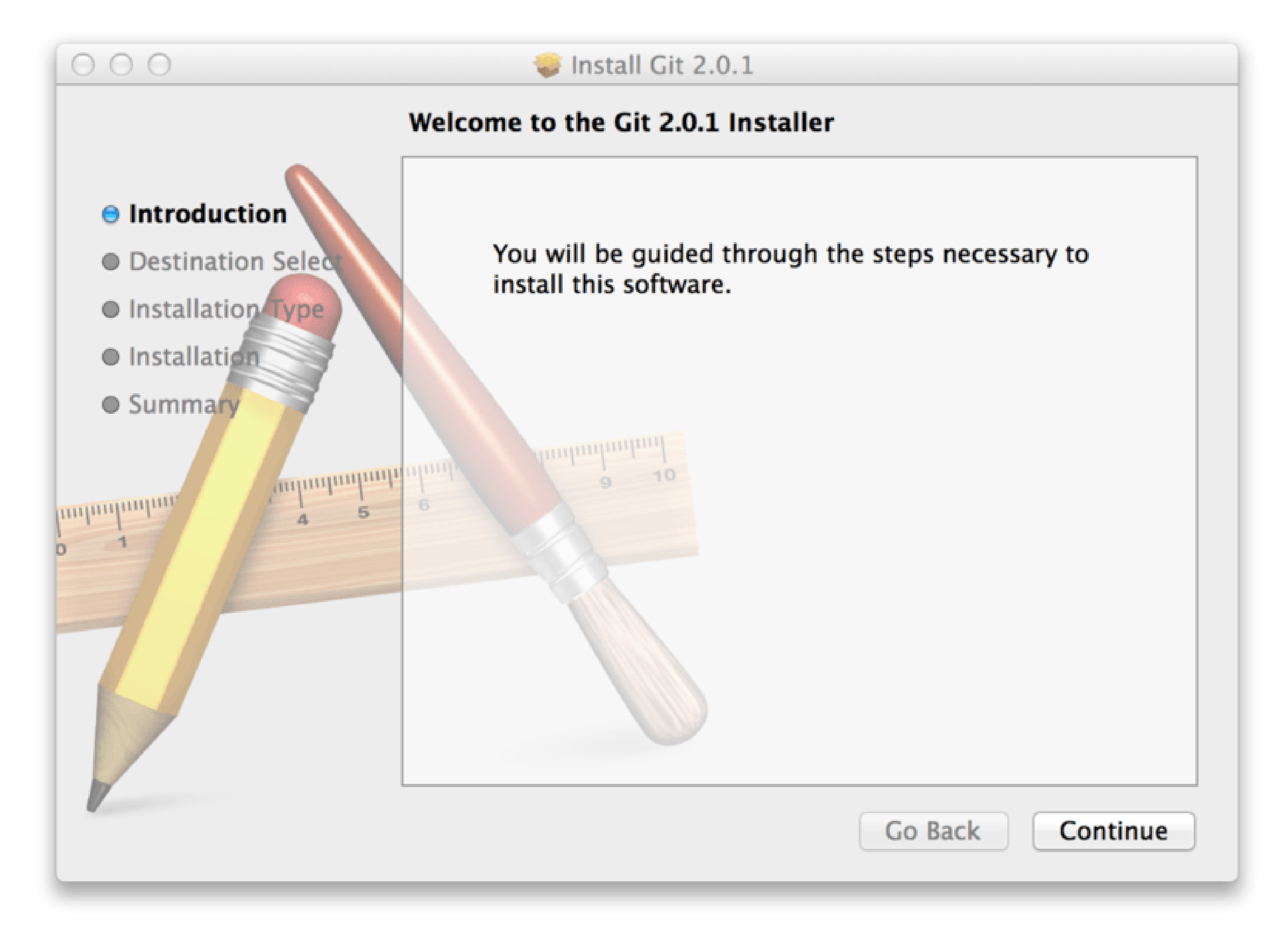


Figure 7. Instalador de Git en OS X.

También puedes instalarlo como parte del instalador de Github para Mac. Su interfaz gráfica de usuario tiene la opción de instalar las herramientas de línea de comandos. Puedes descargar esa herramienta desde el sitio web de Github para Mac en http://mac.github.com.

Instalación en Windows

También hay varias maneras de instalar Git en Windows. La forma más oficial está disponible para ser descargada en el sitio web de Git. Solo tienes que visitar http://git-scm.com/download/win y la descarga empezará automáticamente. Fíjate que éste es un proyecto conocido como Git para Windows (también llamado msysGit), el cual es diferente de Git. Para más información acerca de este proyecto visita http://msysgit.github.io/.

Otra forma de obtener Git fácilmente es mediante la instalación de GitHub para Windows. El instalador incluye la versión de línea de comandos y la interfaz de usuario de Git. Además funciona bien con Powershell y establece correctamente "caching" de credenciales y configuración CRLF adecuada. Aprenderemos acerca de todas estas cosas un poco más adelante, pero por ahora es suficiente mencionar que éstas son cosas que deseas. Puedes descargar este instalador del sitio web de GitHub para Windows en http://windows.github.com.

Instalación a partir del Código Fuente

Algunas personas desean instalar Git a partir de su código fuente debido a que obtendrás una versión más reciente. Los instaladores binarios tienen a estar un poco atrasados. Sin embargo, esto ha hecho muy poca diferencia a medida que Git ha madurado en los últimos años.

Para instalar Git desde el código fuente necesitas tener las siguientes librerías de las que Git depende: curl, zlib, openssl, expat y libiconv. Por ejemplo, si estás en un sistema que tiene yum (como Fedora) o apt-get (como un sistema basado en Debian), puedes usar estos comandos para instalar todas las dependencias:

```
$ yum install curl-devel expat-devel gettext-devel \
  openssl-devel zlib-devel
```

```
$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
  libz-dev libssl-dev
```

Cuando tengas todas las dependencias necesarias, puedes descargar la versión más reciente de Git en diferentes sitios. Puedes obtenerlo a partir del sitio Kernel.org en https://www.kernel.org/pub/software/scm/git, o su "mirror" en el sitio web de GitHub en https://github.com/git/git/releases. Generalmente la más reciente versión en la página web de GitHub es un poco mejor, pero la página de kernel.org también tiene ediciones con firma en caso de que desees verificar tu descarga.

Luego tienes que compilar e instalar de la siguiente manera:

```
$ tar -zxf git-2.0.0.tar.gz
$ cd git-2.0.0
$ make configure
$ ./configure --prefix=/usr
$ make all doc info
$ sudo make install install-doc install-html install-info
```

Una vez hecho esto, también puedes obtener Git, a través del propio Git, para futuras actualizaciones:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

Configurando Git por primera vez

Ahora que tienes Git en tu sistema, vas a querer hacer algunas cosas para personalizar tu entorno de Git. Es necesario hacer estas cosas solamente una vez en tu computadora, y se mantendrán entre actualizaciones. También puedes cambiarlas en cualquier momento volviendo a ejecutar los comandos correspondientes.

Git trae una herramienta llamada `git config` que te permite obtener y establecer variables de configuración que controlan el aspecto y funcionamiento de Git. Estas variables pueden almacenarse en tres sitios distintos:

1. Archivo `/etc/gitconfig`: Contiene valores para todos los usuarios del sistema y todos sus repositorios. Si pasas la opción `--system` a `git config`, lee y escribe específicamente en este archivo.
2. Archivo `~/.gitconfig` o `~/.config/git/config`: Este archivo es específico a tu usuario. Puedes hacer que Git lea y escriba específicamente en este archivo pasando la opción `--global`.
3. Archivo `config` en el directorio de Git (es decir, `.git/config`) del repositorio que estés utilizando actualmente: Este archivo es específico al repositorio actual.

Cada nivel sobrescribe los valores del nivel anterior, por lo que los valores de `.git/config` tienen preferencia sobre los de `/etc/gitconfig`.

En sistemas Windows, Git busca el archivo `.gitconfig` en el directorio `$HOME` (para mucha gente será (`C:\Users\$USER`). También busca el archivo `/etc/gitconfig`, aunque esta ruta es relativa a la raíz MSys, que es donde decidiste instalar Git en tu sistema Windows cuando ejecutaste el instalador.

Tu Identidad

Lo primero que deberás hacer cuando instales Git es establecer tu nombre de usuario y dirección de correo electrónico. Esto es importante porque los "commits" de Git usan esta información, y es introducida de manera inmutable en los commits que envías:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

De nuevo, solo necesitas hacer esto una vez si especificas la opción `--global`, ya que Git siempre usará esta información para todo lo que hagas en ese sistema. Si quieres sobrescribir esta información con otro nombre o dirección de correo para proyectos específicos, puedes ejecutar el comando sin la opción `--global` cuando estés en ese proyecto.

Muchas de las herramientas de interfaz gráfica te ayudarán a hacer esto la primera vez que las uses.

Tu Editor

Ahora que tu identidad está configurada, puedes elegir el editor de texto por defecto que se utilizará cuando Git necesite que introduzcas un mensaje. Si no indicas nada, Git usa el editor por defecto de tu sistema, que generalmente es Vim. Si quieres usar otro editor de texto como Emacs, puedes hacer lo siguiente:

```
$ git config --global core.editor emacs
```

NOTE Vim y Emacs son editores de texto frecuentemente usados por desarrolladores en sistemas basados en Unix como Linux y Mac. Si no estás familiarizado con ninguno de estos editores o estás en un sistema Windows, es posible que necesites buscar instrucciones acerca de cómo configurar tu editor favorito con Git. Si no configuras un editor así y no conoces acerca de Vim o Emacs, es muy factible que termines en un estado bastante confuso en el momento en que sean ejecutados.

Comprobando tu Configuración

Si quieres comprobar tu configuración, puedes usar el comando `git config --list` para mostrar todas las propiedades que Git ha configurado:

```
$ git config --list
user.name=John Doe
user.email=johndoe@example.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

Puede que veas claves repetidas, porque Git lee la misma clave de distintos archivos (`/etc/gitconfig` y `~/.gitconfig`, por ejemplo). En ese caso, Git usa el último valor para cada clave única que ve.

También puedes comprobar qué valor que Git utilizará para una clave específica ejecutando `git`

`config <key>`:

```
$ git config user.name
John Doe
```

¿Cómo obtener ayuda?

Si alguna vez necesitas ayuda usando Git, existen tres formas de ver la página del manual (manpage) para cualquier comando de Git:

```
$ git help <verb>
$ git <verb> --help
$ man git-<verb>
```

Por ejemplo, puedes ver la página del manual para el comando config ejecutando

```
$ git help config
```

Estos comandos son muy útiles porque puedes acceder a ellos desde cualquier sitio, incluso sin conexión. Si las páginas del manual y este libro no son suficientes y necesitas que te ayude una persona, puedes probar en los canales #git o #github del servidor de IRC Freenode (irc.freenode.net). Estos canales están llenos de cientos de personas que conocen muy bien Git y suelen estar dispuestos a ayudar.

Resumen

En este momento debes tener una comprensión básica de lo que es Git, y en que se diferencia de cualquier otro sistema de control de versiones centralizado que pudieras haber utilizado previamente. De igual manera, Git debe estar funcionando en tu sistema y configurado con tu identidad personal. Es hora de aprender los fundamentos de Git.

Fundamentos de Git

Si pudieras leer solo un capítulo para empezar a trabajar con Git, este es el capítulo que debes leer. Este capítulo cubre todos los comandos básicos que necesitas para hacer la gran mayoría de cosas a las que eventualmente vas a dedicar tu tiempo mientras trabajas con Git. Al final del capítulo, deberás ser capaz de configurar e inicializar un repositorio, comenzar y detener el seguimiento de archivos, y preparar (stage) y confirmar (commit) cambios. También te enseñaremos a configurar Git para que ignore ciertos archivos y patrones, cómo enmendar errores rápida y fácilmente, cómo navegar por la historia de tu proyecto y ver cambios entre confirmaciones, y cómo enviar (push) y recibir (pull) de repositorios remotos.

Obteniendo un repositorio Git

Puedes obtener un proyecto Git de dos maneras. La primera es tomar un proyecto o directorio existente e importarlo en Git. La segunda es clonar un repositorio existente en Git desde otro servidor.

Inicializando un repositorio en un directorio existente

Si estás empezando a seguir un proyecto existente en Git, debes ir al directorio del proyecto y usar el siguiente comando:

```
$ git init
```

Esto crea un subdirectorio nuevo llamado `.git`, el cual contiene todos los archivos necesarios del repositorio – un esqueleto de un repositorio de Git. Todavía no hay nada en tu proyecto que esté bajo seguimiento. Puedes revisar Los entresijos internos de Git para obtener más información acerca de los archivos presentes en el directorio `.git` que acaba de ser creado.

Si deseas empezar a controlar versiones de archivos existentes (a diferencia de un directorio vacío), probablemente deberías comenzar el seguimiento de esos archivos y hacer una confirmación inicial. Puedes conseguirlo con unos pocos comandos `git add` para especificar qué archivos quieres controlar, seguidos de un `git commit` para confirmar los cambios:

```
$ git add *.c
$ git add LICENSE
$ git commit -m 'initial project version'
```

Veremos lo que hacen estos comandos más adelante. En este momento, tienes un repositorio de Git con archivos bajo seguimiento y una confirmación inicial.

Clonando un repositorio existente

Si deseas obtener una copia de un repositorio Git existente — por ejemplo, un proyecto en el que te gustaría contribuir — el comando que necesitas es `git clone`. Si estás familizarizado con otros sistemas de control de versiones como Subversion, verás que el comando es "clone" en vez de

"checkout". Es una distinción importante, ya que Git recibe una copia de casi todos los datos que tiene el servidor. Cada versión de cada archivo de la historia del proyecto es descargada por defecto cuando ejecutas `git clone`. De hecho, si el disco de tu servidor se corrompe, puedes usar cualquiera de los clones en cualquiera de los clientes para devolver al servidor al estado en el que estaba cuando fue clonado (puede que pierdas algunos hooks del lado del servidor y demás, pero toda la información acerca de las versiones estará ahí) — véase Configurando Git en un servidor para más detalles.

Puedes clonar un repositorio con `git clone [url]`. Por ejemplo, si quieres clonar la librería de Git llamada libgit2 puedes hacer algo así:

```
$ git clone https://github.com/libgit2/libgit2
```

Esto crea un directorio llamado `libgit2`, inicializa un directorio `.git` en su interior, descarga toda la información de ese repositorio y saca una copia de trabajo de la última versión. Si te metes en el directorio `libgit2`, verás que están los archivos del proyecto listos para ser utilizados. Si quieres clonar el repositorio a un directorio con otro nombre que no sea `libgit2`, puedes especificarlo con la siguiente opción de línea de comandos:

```
$ git clone https://github.com/libgit2/libgit2 mylibgit
```

Ese comando hace lo mismo que el anterior, pero el directorio de destino se llamará `mylibgit`.

Git te permite usar distintos protocolos de transferencia. El ejemplo anterior usa el protocolo `https://`, pero también puedes utilizar `git://` o `usuario@servidor:ruta/del/repositorio.git` que utiliza el protocolo de transferencia SSH. En Configurando Git en un servidor se explicarán todas las opciones disponibles a la hora de configurar el acceso a tu repositorio de Git, y las ventajas e inconvenientes de cada una.

Guardando cambios en el Repositorio

Ya tienes un repositorio Git y un *checkout* o copia de trabajo de los archivos de dicho proyecto. El siguiente paso es realizar algunos cambios y confirmar instantáneas de esos cambios en el repositorio cada vez que el proyecto alcance un estado que quieras conservar.

Recuerda que cada archivo de tu repositorio puede tener dos estados: rastreados y sin rastrear. Los archivos rastreados (*tracked files* en inglés) son todos aquellos archivos que estaban en la última instantánea del proyecto; pueden ser archivos sin modificar, modificados o preparados. Los archivos sin rastrear son todos los demás - cualquier otro archivo en tu directorio de trabajo que no estaba en tu última instantánea y que no están en el área de preparación (*staging area*). Cuando clonas por primera vez un repositorio, todos tus archivos estarán rastreados y sin modificar pues acabas de sacarlos y aun no han sido editados.

Mientras editas archivos, Git los ve como modificados, pues han sido cambiados desde su último *commit*. Luego preparas estos archivos modificados y finalmente confirmas todos los cambios preparados, y repites el ciclo.

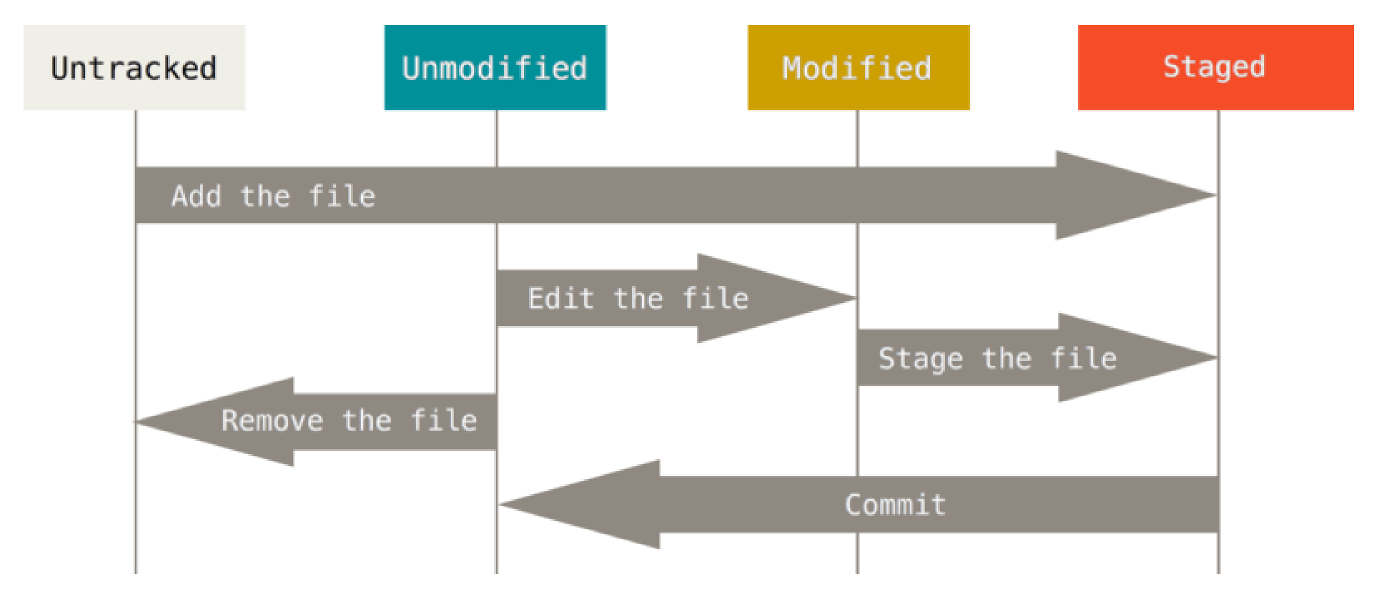


Figure 8. El ciclo de vida del estado de tus archivos.

Revisando el Estado de tus Archivos

La herramienta principal para determinar qué archivos están en qué estado es el comando `git status`. Si ejecutas este comando inmediatamente después de clonar un repositorio, deberías ver algo como esto:

```
$ git status
On branch master
nothing to commit, working directory clean
```

Esto significa que tienes un directorio de trabajo limpio - en otras palabras, que no hay archivos rastreados y modificados. Además, Git no encuentra ningún archivo sin rastrear, de lo contrario aparecerían listados aquí. Finalmente, el comando te indica en cuál rama estás y te informa que no ha variado con respecto a la misma rama en el servidor. Por ahora, la rama siempre será “master”, que es la rama por defecto; no le prestaremos atención ahora. Ramificaciones en Git tratará en detalle las ramas y las referencias.

Supongamos que añades un nuevo archivo a tu proyecto, un simple README. Si el archivo no existía antes, y ejecutas `git status`, verás el archivo sin rastrear de la siguiente manera:

```
$ echo 'My Project' > README
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    README

nothing added to commit but untracked files present (use "git add" to track)
```

Puedes ver que el archivo README está sin rastrear porque aparece debajo del encabezado “Untracked files” (“Archivos no rastreados” en inglés) en la salida. Sin rastrear significa que Git ve

archivos que no tenías en el *commit* anterior. Git no los incluirá en tu próximo *commit* a menos que se lo indiques explícitamente. Se comporta así para evitar incluir accidentalmente archivos binarios o cualquier otro archivo que no quieras incluir. Como tú sí quieres incluir README, debes comenzar a rastrearlo.

Rastrear Archivos Nuevos

Para comenzar a rastrear un archivo debes usar el comando `git add`. Para comenzar a rastrear el archivo README, puedes ejecutar lo siguiente:

```
$ git add README
```

Ahora si vuelves a ver el estado del proyecto, verás que el archivo README está siendo rastreado y está preparado para ser confirmado:

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
```

Puedes ver que está siendo rastreado porque aparece luego del encabezado "Cambios a ser confirmados" ("Changes to be committed" en inglés). Si confirmas en este punto, se guardará en el historial la versión del archivo correspondiente al instante en que ejecutaste `git add`. Anteriormente cuando ejecutaste `git init`, ejecutaste luego `git add (files)` - lo cual inició el rastreo de archivos en tu directorio. El comando `git add` puede recibir tanto una ruta de archivo como de un directorio; si es de un directorio, el comando añade recursivamente los archivos que están dentro de él.

Preparar Archivos Modificados

Vamos a cambiar un archivo que esté rastreado. Si cambias el archivo rastreado llamado "CONTRIBUTING.md" y luego ejecutas el comando `git status`, verás algo parecido a esto:

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

El archivo “CONTRIBUTING.md” aparece en una sección llamada “Changes not staged for commit” (“Cambios no preparado para confirmar” en inglés) - lo que significa que existe un archivo rastreado que ha sido modificado en el directorio de trabajo pero que aun no está preparado. Para prepararlo, ejecutas el comando `git add`. `git add` es un comando que cumple varios propósitos - lo usas para empezar a rastrear archivos nuevos, preparar archivos, y hacer otras cosas como marcar como resuelto archivos en conflicto por combinación. Es más útil que lo veas como un comando para “añadir este contenido a la próxima confirmación” mas que para “añadir este archivo al proyecto”. Ejecutemos `git add` para preparar el archivo “CONTRIBUTING.md” y luego ejecutemos `git status`:

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md
```

Ambos archivos están preparados y formarán parte de tu próxima confirmación. En este momento, supongamos que recuerdas que debes hacer un pequeño cambio en `CONTRIBUTING.md` antes de confirmarlo. Abres de nuevo el archivo, lo cambias y ahora estás listos para confirmar. Sin embargo, ejecutemos `git status` una vez más:

```
$ vim CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

¡¿Pero qué...?! Ahora `CONTRIBUTING.md` aparece como preparado y como no preparado. ¿Cómo es posible? Resulta que Git prepara un archivo de acuerdo al estado que tenía cuando ejecutas el comando `git add`. Si confirmas ahora, se confirmará la versión de `CONTRIBUTING.md` que tenías la última vez que ejecutaste `git add` y no la versión que ves ahora en tu directorio de trabajo al ejecutar `git commit`. Si modificas un archivo luego de ejecutar `git add`, deberás ejecutar `git add` de nuevo para preparar la última versión del archivo:

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md
```

Estatus Abreviado

Si bien es cierto que la salida de `git status` es bastante explícita, también es verdad que es muy extensa. Git ofrece una opción para obtener un estatus abreviado, de manera que puedas ver tus cambios de una forma más compacta. Si ejecutas `git status -s` o `git status --short` obtendrás una salida mucho más simplificada.

```
$ git status -s
 M README
MM Rakefile
A  lib/git.rb
M  lib/simplegit.rb
?? LICENSE.txt
```

Los archivos nuevos que no están rastreados tienen un `??` a su lado, los archivos que están

preparados tienen una A y los modificados una M. El estado aparece en dos columnas - la columna de la izquierda indica el estado preparado y la columna de la derecha indica el estado sin preparar. Por ejemplo, en esa salida, el archivo README está modificado en el directorio de trabajo pero no está preparado, mientras que lib/simplegit.rb está modificado y preparado. El archivo Rakefile fue modificado, preparado y modificado otra vez por lo que existen cambios preparados y sin preparar.

Ignorar Archivos

A veces, tendrás algún tipo de archivo que no quieres que Git añada automáticamente o más aun, que ni siquiera quieras que aparezca como no rastreado. Este suele ser el caso de archivos generados automáticamente como trazas o archivos creados por tu sistema de construcción. En estos casos, puedes crear un archivo llamado .gitignore que liste patrones a considerar. Este es un ejemplo de un archivo .gitignore:

```
$ cat .gitignore
*.[oa]
*~
```

La primera línea le indica a Git que ignore cualquier archivo que termine en ".o" o ".a" - archivos de objeto o librerías que pueden ser producto de compilar tu código. La segunda línea le indica a Git que ignore todos los archivos que termine con una tilde (~), lo cual es usado por varios editores de texto como Emacs para marcar archivos temporales. También puedes incluir cosas como trazas, temporales, o pid directamente; documentación generada automáticamente; etc. Crear un archivo .gitignore antes de comenzar a trabajar es generalmente una buena idea pues así evitas confirmar accidentalmente archivos que en realidad no quieres incluir en tu repositorio Git.

Las reglas sobre los patrones que puedes incluir en el archivo .gitignore son las siguientes:

- Ignorar las líneas en blanco y aquellas que comiencen con #.
- Aceptar patrones glob estándar.
- Los patrones pueden terminar en barra (/) para especificar un directorio.
- Los patrones pueden negarse si se añade al principio el signo de exclamación (!).

Los patrones glob son una especia de expresión regular simplificada usada por los terminales. Un asterisco (*) corresponde a cero o más caracteres; [abc] corresponde a cualquier carácter dentro de los corchetes (en este caso a, b o c); el signo de interrogación (?) corresponde a un carácter cualquier; y los corchetes sobre caracteres separados por un guión ([0-9]) corresponde a cualquier carácter entre ellos (en este caso del 0 al 9). También puedes usar dos asteriscos para indicar directorios anidados; a/**/z coincide con a/z, a/b/z, a/b/c/z, etc.

Aquí puedes ver otro ejemplo de un archivo .gitignore:

```
# ignora los archivos terminados en .a
*.a

# pero no lib.a, aun cuando había ignorado los archivos terminados en .a en la linea
anterior
!lib.a

# ignora unicamente el archivo TODO de la raiz, no subdir/TODO
/TODO

# ignora todos los archivos del directorio build/
build/

# ignora doc/notes.txt, pero este no doc/server/arch.txt
doc/*.txt

# ignora todos los archivos .txt el directorio doc/
doc/**/*.txt
```

TIP GitHub mantiene una extensa lista de archivos `.gitignore` adecuados a docenas de proyectos y lenguajes en https://github.com/github/gitignore en caso de que quieras tener un punto de partida para tu proyecto.

Ver los Cambios Preparados y No Preparados

Si el comando `git status` es muy impreciso para ti - quieres ver exactamente que ha cambiado, no solo cuáles archivos lo han hecho - puedes usar el comando `git diff`. Hablaremos sobre `git diff` más adelante, pero lo usarás probablemente para responder estas dos preguntas: ¿Qué has cambiado pero aun no has preparado? y ¿Qué has preparado y está listo para confirmar? A pesar de que `git status` responde a estas preguntas de forma muy general listando el nombre de los archivos, `git diff` te muestra las líneas exactas que fueron añadidas y eliminadas, es decir, el parche.

Supongamos que editas y preparas el archivo `README` de nuevo y luego editas `CONTRIBUTING.md` pero no lo preparas. Si ejecutas el comando `git status`, verás algo como esto:

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

Para ver qué has cambiado pero aun no has preparado, escribe `git diff` sin más parámetros:

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
 Please include a nice description of your changes when you submit your PR;
 if we have to read the whole diff to figure out why you're contributing
 in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if you patch is
+longer than a dozen lines.

 If you are starting to work on a particular area, feel free to submit a PR
 that highlights your work in progress (and note in the PR title that it's
```

Este comando compara lo que tienes en tu directorio de trabajo con lo que está en el área de preparación. El resultado te indica los cambios que has hecho pero que aun no has preparado.

Si quieres ver lo que has preparado y será incluido en la próxima confirmación, puedes usar `git diff --staged`. Este comando compara tus cambios preparados con la última instantánea confirmada.

```
$ git diff --staged
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+My Project
```

Es importante resaltar que al llamar a `git diff` sin parámetros no verás los cambios desde tu última confirmación - solo verás los cambios que aun no están preparados. Esto puede ser confuso porque si preparas todos tus cambios, `git diff` no te devolverá ninguna salida.

Pasemos a otro ejemplo, si preparas el archivo `CONTRIBUTING.md` y luego lo editas, puedes usar `git diff` para ver los cambios en el archivo que están preparados y los cambios que no lo están. Si nuestro ambiente es como este:

```
$ git add CONTRIBUTING.md
$ echo 'test line' >> CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   CONTRIBUTING.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

Puedes usar `git diff` para ver qué está sin preparar

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 643e24f..87f08c8 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -119,3 +119,4 @@ at the
 ## Starter Projects

 See our [projects
list](https://github.com/libgit2/libgit2/blob/development/PROJECTS.md).
+# test line
```

y `git diff --cached` para ver que has preparado hasta ahora (--staged y --cached son sinónimos):

```
$ git diff --cached
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
 Please include a nice description of your changes when you submit your PR;
 if we have to read the whole diff to figure out why you're contributing
 in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if you patch is
+longer than a dozen lines.

 If you are starting to work on a particular area, feel free to submit a PR
 that highlights your work in progress (and note in the PR title that it's
```

> **NOTE**
>
> *Git Diff como Herramienta Externa*
>
> A lo largo del libro, continuaremos usando el comando `git diff` de distintas maneras. Existe otra forma de ver estas diferencias si prefieres utilizar una interfaz gráfica u otro programa externo. Si ejecutas `git difftool` en vez de `git diff`, podrás ver los cambios con programas de este tipo como Araxis, emerge, vimdiff y más. Ejecuta `git difftool --tool-help` para ver qué tienes disponible en tu sistema.

Confirmar tus Cambios

Ahora que tu área de preparación está como quieres, puedes confirmar tus cambios. Recuerda que cualquier cosa que no esté preparada - cualquier archivo que hayas creado o modificado y que no hayas agregado con `git add` desde su edición - no será confirmado. Se mantendrán como archivos modificados en tu disco. En este caso, digamos que la última vez que ejecutaste `git status` verificaste que todo estaba preparado y que estás listos para confirmar tus cambios. La forma más sencilla de confirmar es escribiendo `git commit`:

```
$ git commit
```

Al hacerlo, arrancará el editor de tu preferencia. (El editor se establece a través de la variable de ambiente `$EDITOR` de tu terminal - usualmente es vim o emacs, aunque puedes configurarlo con el editor que quieras usando el comando `git config --global core.editor` tal como viste en Inicio - Sobre el Control de Versiones).

El editor mostrará el siguiente texto (este ejemplo corresponde a una pantalla de Vim):

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   new file:   README
#   modified:   CONTRIBUTING.md
#
~
~
~
".git/COMMIT_EDITMSG" 9L, 283C
```

Puedes ver que el mensaje de confirmación por defecto contiene la última salida del comando `git status` comentada y una línea vacía encima de ella. Puedes eliminar estos comentarios y escribir tu mensaje de confirmación, o puedes dejarlos allí para ayudarte a recordar qué estás confirmando. (Para obtener una forma más explícita de recordar qué has modificado, puedes pasar la opción `-v` a `git commit`. Al hacerlo se incluirá en el editor el diff de tus cambios para que veas exactamente qué cambios estás confirmando.) Cuando sales del editor, Git crea tu confirmación con tu mensaje (eliminando el texto comentado y el diff).

Otra alternativa es escribir el mensaje de confirmación directamente en el comando `commit` utilizando la opción -m:

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master 463dc4f] Story 182: Fix benchmarks for speed
 2 files changed, 2 insertions(+)
 create mode 100644 README
```

¡Has creado tu primera confirmación (o *commit*)! Puedes ver que la confirmación te devuelve una salida descriptiva: indica cuál rama has confirmado (`master`), que *checksum* SHA-1 tiene el *commit* (`463dc4f`), cuántos archivos han cambiado y estadísticas sobre las líneas añadidas y eliminadas en el *commit*.

Recuerda que la confirmación guarda una instantánea de tu área de preparación. Todo lo que no hayas preparado sigue allí modificado; puedes hacer una nueva confirmación para añadirlo a tu historial. Cada vez que realizas un *commit*, guardas una instantánea de tu proyecto la cual puedes usar para comparar o volver a ella luego.

Saltar el Área de Preparación

A pesar de que puede resultar muy útil para ajustar los *commits* tal como quieres, el área de preparación es a veces un paso más complejo a lo que necesitas para tu flujo de trabajo. Si quieres saltarte el área de preparación, Git te ofrece un atajo sencillo. Añadiendo la opción `-a` al comando `git commit` harás que Git prepare automáticamente todos los archivos rastreados antes de confirmarlos, ahorrándote el paso de `git add`:

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md

no changes added to commit (use "git add" and/or "git commit -a")
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
 1 file changed, 5 insertions(+), 0 deletions(-)
```

Fíjate que en este caso no fue necesario ejecutar `git add` sobre el archivo `CONTRIBUTING.md` antes de confirmar.

Eliminar Archivos

Para eliminar archivos de Git, debes eliminarlos de tus archivos rastreados (o mejor dicho, eliminarlos del área de preparación) y luego confirmar. Para ello existe el comando `git rm`, que además elimina el archivo de tu directorio de trabajo de manera que no aparezca la próxima vez como un archivo no rastreado.

Si simplemente eliminas el archivo de tu directorio de trabajo, aparecerá en la sección “Changes not staged for commit” (esto es, *sin preparar*) en la salida de `git status`:

```
$ rm PROJECTS.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    PROJECTS.md

no changes added to commit (use "git add" and/or "git commit -a")
```

Ahora, si ejecutas `git rm`, entonces se prepara la eliminación del archivo:

```
$ git rm PROJECTS.md
rm 'PROJECTS.md'
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    deleted:    PROJECTS.md
```

Con la próxima confirmación, el archivo habrá desaparecido y no volverá a ser rastreado. Si modificaste el archivo y ya lo habías añadido al índice, tendrás que forzar su eliminación con la opción `-f`. Esta propiedad existe por seguridad, para prevenir que elimines accidentalmente datos que aun no han sido guardados como una instantánea y que por lo tanto no podrás recuperar luego con Git.

Otra cosa que puedas querer hacer es mantener el archivo en tu directorio de trabajo pero eliminarlo del área de preparación. En otras palabras, quisieras mantener el archivo en tu disco duro pero sin que Git lo siga rastreando. Esto puede ser particularmente útil si olvidaste añadir algo en tu archivo `.gitignore` y lo preparaste accidentalmente, algo como un gran archivo de trazas a un montón de archivos compilados `.a`. Para hacerlo, utiliza la opción `--cached`:

```
$ git rm --cached README
```

Al comando `git rm` puedes pasarle archivos, directorios y patrones glob. Lo que significa que puedes hacer cosas como

```
$ git rm log/\*.log
```

Fíjate en la barra invertida (`\`) antes del asterisco `*`. Esto es necesario porque Git hace su propia expansión de nombres de archivo, aparte de la expansión hecha por tu terminal. Este comando elimina todos los archivo que tengan la extensión `.log` dentro del directorio `log/`. O también puedes hacer algo como:

```
$ git rm \*~
```

Este comando elimina todos los archivos que acaben con `~`.

Cambiar el Nombre de los Archivos

Al contrario que muchos sistemas VCS, Git no rastrea explícitamente los cambios de nombre en archivos. Si renombras un archivo en Git, no se guardará ningún metadato que indique que renombraste el archivo. Sin embargo, Git es bastante listo como para detectar estos cambios luego que los has hecho - más adelante, veremos cómo se detecta el cambio de nombre.

Por esto, resulta confuso que Git tenga un comando `mv`. Si quieres renombrar un archivo en Git,

puedes ejecutar algo como

```
$ git mv file_from file_to
```

y funcionará bien. De hecho, si ejecutas algo como eso y ves el estatus, verás que Git lo considera como un renombramiento de archivo:

```
$ git mv README.md README
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
```

Sin embargo, eso es equivalente a ejecutar algo como esto:

```
$ mv README.md README
$ git rm README.md
$ git add README
```

Git se da cuenta que es un renombramiento implícito, así que no importa si renombras el archivo de esa manera o a través del comando `mv`. La única diferencia real es que `mv` es un solo comando en vez de tres - existe por conveniencia. De hecho, puedes usar la herramienta que quieras para renombrar un archivo y luego realizar el proceso rm/add antes de confirmar.

Ver el Historial de Confirmaciones

Después de haber hecho varias confirmaciones, o si has clonado un repositorio que ya tenía un histórico de confirmaciones, probablemente quieras mirar atrás para ver qué modificaciones se han llevado a cabo. La herramienta más básica y potente para hacer esto es el comando `git log`.

Estos ejemplos usan un proyecto muy sencillo llamado “simplegit”. Para clonar el proyecto, ejecuta:

```
git clone https://github.com/schacon/simplegit-progit
```

Cuando ejecutes `git log` sobre este proyecto, deberías ver una salida similar a esta:

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
```

Por defecto, si no pasas ningún parámetro, `git log` lista las confirmaciones hechas sobre ese repositorio en orden cronológico inverso. Es decir, las confirmaciones más recientes se muestran al principio. Como puedes ver, este comando lista cada confirmación con su suma de comprobación SHA-1, el nombre y dirección de correo del autor, la fecha y el mensaje de confirmación.

El comando `git log` proporciona gran cantidad de opciones para mostrarte exactamente lo que buscas. Aquí veremos algunas de las más usadas.

Una de las opciones más útiles es `-p`, que muestra las diferencias introducidas en cada confirmación. También puedes usar la opción `-2`, que hace que se muestren únicamente las dos últimas entradas del historial:

```
$ git log -p -2
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -5,7 +5,7 @@ require 'rake/gempackagetask'
 spec = Gem::Specification.new do |s|
     s.platform  =   Gem::Platform::RUBY
     s.name      =   "simplegit"
-    s.version   =   "0.1.0"
+    s.version   =   "0.1.1"
     s.author    =   "Scott Chacon"
     s.email     =   "schacon@gee-mail.com"
     s.summary   =   "A simple gem for using Git in Ruby code."

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index a0a60ae..47c6340 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -18,8 +18,3 @@ class SimpleGit
     end

 end
-
-if $0 == __FILE__
-  git = SimpleGit.new
-  puts git.show
-end
\ No newline at end of file
```

Esta opción muestra la misma información, pero añadiendo tras cada entrada las diferencias que le corresponden. Esto resulta muy útil para revisiones de código, o para visualizar rápidamente lo que ha pasado en las confirmaciones enviadas por un colaborador. También puedes usar con `git log` una serie de opciones de resumen. Por ejemplo, si quieres ver algunas estadísticas de cada confirmación, puedes usar la opción `--stat`:

```
$ git log --stat
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

 Rakefile | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

 lib/simplegit.rb | 5 -----
 1 file changed, 5 deletions(-)

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit

 README           |  6 ++++++
 Rakefile         | 23 +++++++++++++++++++++++
 lib/simplegit.rb | 25 +++++++++++++++++++++++++
 3 files changed, 54 insertions(+)
```

Como puedes ver, la opción `--stat` imprime tras cada confirmación una lista de archivos modificados, indicando cuántos han sido modificados y cuántas líneas han sido añadidas y eliminadas para cada uno de ellos, y un resumen de toda esta información.

Otra opción realmente útil es `--pretty`, que modifica el formato de la salida. Tienes unos cuantos estilos disponibles. La opción `oneline` imprime cada confirmación en una única línea, lo que puede resultar útil si estás analizando gran cantidad de confirmaciones. Otras opciones son `short`, `full` y `fuller`, que muestran la salida en un formato parecido, pero añadiendo menos o más información, respectivamente:

```
$ git log --pretty=oneline
ca82a6dff817ec66f44342007202690a93763949 changed the version number
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test
a11bef06a3f659402fe7563abf99ad00de2209e6 first commit
```

La opción más interesante es `format`, que te permite especificar tu propio formato. Esto resulta especialmente útil si estás generando una salida para que sea analizada por otro programa —como especificas el formato explícitamente, sabes que no cambiará en futuras actualizaciones de Git—:

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 6 years ago : changed the version number
085bb3b - Scott Chacon, 6 years ago : removed unnecessary test
a11bef0 - Scott Chacon, 6 years ago : first commit
```

Opciones útiles de `git log --pretty=format` lista algunas de las opciones más útiles aceptadas por `format`.

Table 1. Opciones útiles de `git log --pretty=format`

Opción	Descripción de la salida
`%H`	Hash de la confirmación
`%h`	Hash de la confirmación abreviado
`%T`	Hash del árbol
`%t`	Hash del árbol abreviado
`%P`	Hashes de las confirmaciones padre
`%p`	Hashes de las confirmaciones padre abreviados
`%an`	Nombre del autor
`%ae`	Dirección de correo del autor
`%ad`	Fecha de autoría (el formato respeta la opción `--date`)
`%ar`	Fecha de autoría, relativa
`%cn`	Nombre del confirmador
`%ce`	Dirección de correo del confirmador
`%cd`	Fecha de confirmación
`%cr`	Fecha de confirmación, relativa
`%s`	Asunto

Puede que te estés preguntando la diferencia entre *autor* (*author*) y *confirmador* (*committer*). El autor es la persona que escribió originalmente el trabajo, mientras que el confirmador es quien lo aplicó. Por tanto, si mandas un parche a un proyecto, y uno de sus miembros lo aplica, ambos recibiréis reconocimiento —tú como autor, y el miembro del proyecto como confirmador—. Veremos esta distinción en mayor profundidad en Git en entornos distribuidos.

Las opciones `oneline` y `format` son especialmente útiles combinadas con otra opción llamada `--graph`. Ésta añade un pequeño gráfico ASCII mostrando tu historial de ramificaciones y uniones:

```
$ git log --pretty=format:"%h %s" --graph
* 2d3acf9 ignore errors from SIGCHLD on trap
*  5e3ee11 Merge branch 'master' of git://github.com/dustin/grit
|\
| * 420eac9 Added a method for getting the current branch.
* | 30e367c timeout code and tests
* | 5a09431 add timeout protection to grit
* | e1193f8 support for heads with slashes in them
|/
* d6016bc require time for xmlschema
*  11d191e Merge branch 'defunkt' into local
```

Este tipo de salidas serán más interesantes cuando empecemos a hablar sobre ramificaciones y combinaciones en el próximo capítulo.

Éstas son sólo algunas de las opciones para formatear la salida de `git log` —existen muchas más. Opciones típicas de `git log` lista las opciones vistas hasta ahora, y algunas otras opciones de formateo que pueden resultarte útiles, así como su efecto sobre la salida.

Table 2. Opciones típicas de `git log`

Opción	Descripción
`-p`	Muestra el parche introducido en cada confirmación.
`--stat`	Muestra estadísticas sobre los archivos modificados en cada confirmación.
`--shortstat`	Muestra solamente la línea de resumen de la opción `--stat`.
`--name-only`	Muestra la lista de archivos afectados.
`--name-status`	Muestra la lista de archivos afectados, indicando además si fueron añadidos, modificados o eliminados.
`--abbrev-commit`	Muestra solamente los primeros caracteres de la suma SHA-1, en vez de los 40 caracteres de que se compone.
`--relative-date`	Muestra la fecha en formato relativo (por ejemplo, “2 weeks ago” (“hace 2 semanas”)) en lugar del formato completo.
`--graph`	Muestra un gráfico ASCII con la historia de ramificaciones y uniones.
`--pretty`	Muestra las confirmaciones usando un formato alternativo. Posibles opciones son oneline, short, full, fuller y format (mediante el cual puedes especificar tu propio formato).

Limitar la Salida del Historial

Además de las opciones de formateo, `git log` acepta una serie de opciones para limitar su salida —es decir, opciones que te permiten mostrar únicamente parte de las confirmaciones—. Ya has visto una de ellas, la opción `-2`, que muestra sólo las dos últimas confirmaciones. De hecho, puedes hacer `-<n>`, siendo `n` cualquier entero, para mostrar las últimas `n` confirmaciones. En realidad es poco probable que uses esto con frecuencia, ya que Git por defecto pagina su salida para que veas cada página del historial por separado.

Sin embargo, las opciones temporales como `--since` (desde) y `--until` (hasta) sí que resultan muy útiles. Por ejemplo, este comando lista todas las confirmaciones hechas durante las dos últimas semanas:

```
$ git log --since=2.weeks
```

Este comando acepta muchos formatos. Puedes indicar una fecha concreta (`"2008-01-15"`), o relativa, como `"2 years 1 day 3 minutes ago"` (`"hace 2 años, 1 día y 3 minutos"`).

También puedes filtrar la lista para que muestre sólo aquellas confirmaciones que cumplen ciertos criterios. La opción `--author` te permite filtrar por autor, y `--grep` te permite buscar palabras clave entre los mensajes de confirmación. (Ten en cuenta que si quieres aplicar ambas opciones simultáneamente, tienes que añadir `--all-match`, o el comando mostrará las confirmaciones que cumplan cualquiera de las dos, no necesariamente las dos a la vez.)

Otra opción útil es `-S`, la cual recibe una cadena y solo muestra las confirmaciones que cambiaron el código añadiendo o eliminando la cadena. Por ejemplo, si quieres encontrar la última confirmación que añadió o eliminó una referencia a una función específica, puede ejecutar:

```
$ git log -Sfunction_name
```

La última opción verdaderamente útil para filtrar la salida de `git log` es especificar una ruta. Si especificas la ruta de un directorio o archivo, puedes limitar la salida a aquellas confirmaciones que introdujeron un cambio en dichos archivos. Ésta debe ser siempre la última opción, y suele ir precedida de dos guiones (`--`) para separar la ruta del resto de opciones.

En Opciones para limitar la salida de `git log` se listan estas opciones, y algunas otras bastante comunes, a modo de referencia.

Table 3. Opciones para limitar la salida de `git log`

Opción	Descripción
`-(n)`	Muestra solamente las últimas n confirmaciones
`--since, --after`	Muestra aquellas confirmaciones hechas después de la fecha especificada.
`--until, --before`	Muestra aquellas confirmaciones hechas antes de la fecha especificada.
`--author`	Muestra solo aquellas confirmaciones cuyo autor coincide con la cadena especificada.
`--committer`	Muestra solo aquellas confirmaciones cuyo confirmador coincide con la cadena especificada.
`-S`	Muestra solo aquellas confirmaciones que añadan o eliminen código que corresponda con la cadena especificada.

Por ejemplo, si quieres ver cuáles de las confirmaciones hechas sobre archivos de prueba del código fuente de Git fueron enviadas por Junio Hamano, y no fueron uniones, en el mes de octubre

de 2008, ejecutarías algo así:

```
$ git log --pretty="%h - %s" --author=gitster --since="2008-10-01" \
   --before="2008-11-01" --no-merges -- t/
5610e3b - Fix testcase failure when extended attributes are in use
acd3b9e - Enhance hold_lock_file_for_{update,append}() API
f563754 - demonstrate breakage of detached checkout with symbolic link HEAD
d1a43f2 - reset --hard/read-tree --reset -u: remove unmerged new paths
51a94af - Fix "checkout --track -b newbranch" on detached HEAD
b0ad11e - pull: allow "git pull origin $something:$current_branch" into an unborn
branch
```

De las casi 40.000 confirmaciones en la historia del código fuente de Git, este comando muestra las 6 que cumplen estas condiciones.

Deshacer Cosas

En cualquier momento puede que quieras deshacer algo. Aquí repasaremos algunas herramientas básicas usadas para deshacer cambios que hayas hecho. Ten cuidado, a veces no es posible recuperar algo luego que lo has deshecho. Esta es una de las pocas áreas en las que Git puede perder parte de tu trabajo si cometes un error.

Uno de las acciones más comunes a deshacer es cuando confirmas un cambio antes de tiempo y olvidas agregar algún archivo, o te equivocas en el mensaje de confirmación. Si quieres rehacer la confirmación, puedes reconfirmar con la opción `--amend`:

```
$ git commit --amend
```

Este comando utiliza tu área de preparación para la confirmación. Si no has hecho cambios desde tu última confirmación (por ejemplo, ejecutas este comando justo después de tu confirmación anterior), entonces la instantánea lucirá exactamente igual, y lo único que cambiarás será el mensaje de confirmación.

Se lanzará el mismo editor de confirmación, pero verás que ya incluye el mensaje de tu confirmación anterior. Puedes editar el mensaje como siempre y se sobreescribirá tu confirmación anterior.

Por ejemplo, si confirmas y luego te das cuenta que olvidaste preparar los cambios de un archivo que querías incluir en esta confirmación, puedes hacer lo siguiente:

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit --amend
```

Al final terminarás con una sola confirmación - la segunda confirmación reemplaza el resultado de la primera.

Deshacer un Archivo Preparado

Las siguientes dos secciones demuestran cómo lidiar con los cambios de tu área de preparación y tú directorio de trabajo. Afortunadamente, el comando que usas para determinar el estado de esas dos áreas también te recuerda cómo deshacer los cambios en ellas. Por ejemplo, supongamos que has cambiado dos archivos y que quieres confirmarlos como dos cambios separados, pero accidentalmente has escrito `git add *` y has preparado ambos. ¿Cómo puedes sacar del área de preparación uno de ellos? El comando `git status` te recuerda cómo:

```
$ git add .
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
    modified:   CONTRIBUTING.md
```

Justo debajo del texto “Changes to be committed” (“Cambios a ser confirmados”, en inglés), verás que dice que uses `git reset HEAD <file>...` para deshacer la preparación. Por lo tanto, usemos el consejo para deshacer la preparación del archivo `CONTRIBUTING.md`:

```
$ git reset HEAD CONTRIBUTING.md
Unstaged changes after reset:
M   CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

El comando es un poco raro, pero funciona. El archivo `CONTRIBUTING.md` esta modificado y, nuevamente, no preparado.

NOTE

A pesar de que `git reset` *puede* ser un comando peligroso si lo llamas con `--hard`, en este caso el archivo que está en tu directorio de trabajo no se toca. Ejecutar `git reset` sin opciones no es peligroso - solo toca el área de preparación.

Por ahora lo único que necesitas saber sobre el comando `git reset` es esta invocación mágica. Entraremos en mucho más detalle sobre qué hace `reset` y como dominarlo para que haga cosas

realmente interesantes en Reiniciar Desmitificado.

Deshacer un Archivo Modificado

¿Qué tal si te das cuenta que no quieres mantener los cambios del archivo `CONTRIBUTING.md`? ¿Cómo puedes restaurarlo fácilmente - volver al estado en el que estaba en la última confirmación (o cuando estaba recién clonado, o como sea que haya llegado a tu directorio de trabajo)? Afortunadamente, `git status` también te dice cómo hacerlo. En la salida anterior, el área no preparada lucía así:

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

Allí se te indica explícitamente como descartar los cambios que has hecho. Hagamos lo que nos dice:

```
$ git checkout -- CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
```

Ahora puedes ver que los cambios se han revertido.

IMPORTANT	Es importante entender que `git checkout -- [archivo]` es un comando peligroso. Cualquier cambio que le hayas hecho a ese archivo desaparecerá - acabas de sobreescribirlo con otro archivo. Nunca utilices este comando a menos que estés absolutamente seguro de que ya no quieres el archivo.

Para mantener los cambios que has hecho y a la vez deshacerte del archivo temporalmente, hablaremos sobre cómo esconder archivos (*stashing*, en inglés) y sobre ramas en Ramificaciones en Git; normalmente, estas son las mejores maneras de hacerlo.

Recuerda, todo lo que esté *confirmado* en Git puede recuperarse. Incluso *commits* que estuvieron en ramas que han sido eliminadas o *commits* que fueron sobreescritos con `--amend` pueden recuperarse (véase Recuperación de datos para recuperación de datos). Sin embargo, es posible que no vuelvas a ver jamás cualquier cosa que pierdas y que nunca haya sido confirmada.

Trabajar con Remotos

Para poder colaborar en cualquier proyecto Git, necesitas saber cómo gestionar repositorios remotos. Los repositorios remotos son versiones de tu proyecto que están hospedadas en Internet

en cualquier otra red. Puedes tener varios de ellos, y en cada uno tendrás generalmente permisos de solo lectura o de lectura y escritura. Colaborar con otras personas implica gestionar estos repositorios remotos y enviar y traer datos de ellos cada vez que necesites compartir tu trabajo. Gestionar repositorios remotos incluye saber cómo añadir un repositorio remoto, eliminar los remotos que ya no son válidos, gestionar varias ramas remotas y definir si deben rastrearse o no, y más. En esta sección, trataremos algunas de estas habilidades de gestión de remotos.

Ver Tus Remotos

Para ver los remotos que tienes configurados, debes ejecutar el comando `git remote`. Mostrará los nombres de cada uno de los remotos que tienes especificados. Si has clonado tu repositorio, deberías ver al menos *origin* (origen, en inglés) - este es el nombre que por defecto Git le da al servidor del que has clonado:

```
$ git clone https://github.com/schacon/ticgit
Cloning into 'ticgit'...
remote: Reusing existing pack: 1857, done.
remote: Total 1857 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (1857/1857), 374.35 KiB | 268.00 KiB/s, done.
Resolving deltas: 100% (772/772), done.
Checking connectivity... done.
$ cd ticgit
$ git remote
origin
```

También puedes pasar la opción `-v`, la cual muestra las URLs que Git ha asociado al nombre y que serán usadas al leer y escribir en ese remoto:

```
$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
```

Si tienes más de un remoto, el comando los listará todos. Por ejemplo, un repositorio con múltiples remotos para trabajar con distintos colaboradores podría verse de la siguiente manera.

```
$ cd grit
$ git remote -v
bakkdoor  https://github.com/bakkdoor/grit (fetch)
bakkdoor  https://github.com/bakkdoor/grit (push)
cho45     https://github.com/cho45/grit (fetch)
cho45     https://github.com/cho45/grit (push)
defunkt   https://github.com/defunkt/grit (fetch)
defunkt   https://github.com/defunkt/grit (push)
koke      git://github.com/koke/grit.git (fetch)
koke      git://github.com/koke/grit.git (push)
origin    git@github.com:mojombo/grit.git (fetch)
origin    git@github.com:mojombo/grit.git (push)
```

Esto significa que podemos traer contribuciones de cualquiera de estos usuarios fácilmente. Es posible que también tengamos permisos para enviar datos a algunos, aunque no podemos saberlo desde aquí.

Fíjate que estos remotos usan distintos protocolos; hablaremos sobre ello más adelante, en Configurando Git en un servidor.

Añadir Repositorios Remotos

En secciones anteriores hemos mencionado y dado alguna demostración de cómo añadir repositorios remotos. Ahora veremos explícitamente cómo hacerlo. Para añadir un remoto nuevo y asociarlo a un nombre que puedas referenciar fácilmente, ejecuta `git remote add [nombre] [url]`:

```
$ git remote
origin
$ git remote add pb https://github.com/paulboone/ticgit
$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
pb  https://github.com/paulboone/ticgit (fetch)
pb  https://github.com/paulboone/ticgit (push)
```

A partir de ahora puedes usar el nombre `pb` en la línea de comandos en lugar de la URL entera. Por ejemplo, si quieres traer toda la información que tiene Paul pero tú aun no tienes en tu repositorio, puedes ejecutar `git fetch pb`:

```
$ git fetch pb
remote: Counting objects: 43, done.
remote: Compressing objects: 100% (36/36), done.
remote: Total 43 (delta 10), reused 31 (delta 5)
Unpacking objects: 100% (43/43), done.
From https://github.com/paulboone/ticgit
 * [new branch]      master     -> pb/master
 * [new branch]      ticgit     -> pb/ticgit
```

La rama maestra de Paul ahora es accesible localmente con el nombre `pb/master` - puedes combinarla con alguna de tus ramas, o puedes crear una rama local en ese punto si quieres inspeccionarla. (Hablaremos con más detalle acerca de qué son las ramas y cómo utilizarlas en Ramificaciones en Git.)

Traer y Combinar Remotos

Como hemos visto hasta ahora, para obtener datos de tus proyectos remotos puedes ejecutar:

```
$ git fetch [remote-name]
```

El comando irá al proyecto remoto y se traerá todos los datos que aun no tienes de dicho remoto. Luego de hacer esto, tendrás referencias a todas las ramas del remoto, las cuales puedes combinar e inspeccionar cuando quieras.

Si clonas un repositorio, el comando de clonar automáticamente añade ese repositorio remoto con el nombre “origin”. Por lo tanto, `git fetch origin` se trae todo el trabajo nuevo que ha sido enviado a ese servidor desde que lo clonaste (o desde la última vez que trajiste datos). Es importante destacar que el comando `git fetch` solo trae datos a tu repositorio local - ni lo combina automáticamente con tu trabajo ni modifica el trabajo que llevas hecho. La combinación con tu trabajo debes hacerla manualmente cuando estés listo.

Si has configurado una rama para que rastree una rama remota (más información en la siguiente sección y en Ramificaciones en Git), puedes usar el comando `git pull` para traer y combinar automáticamente la rama remota con tu rama actual. Es posible que este sea un flujo de trabajo mucho más cómodo y fácil para ti; y por defecto, el comando `git clone` le indica automáticamente a tu rama maestra local que rastree la rama maestra remota (o como se llame la rama por defecto) del servidor del que has clonado. Generalmente, al ejecutar `git pull` traerás datos del servidor del que clonaste originalmente y se intentará combinar automáticamente la información con el código en el que estás trabajando.

Enviar a Tus Remotos

Cuando tienes un proyecto que quieres compartir, debes enviarlo a un servidor. El comando para hacerlo es simple: `git push [nombre-remoto] [nombre-rama]`. Si quieres enviar tu rama `master` a tu servidor `origin` (recuerda, clonar un repositorio establece esos nombres automáticamente), entonces puedes ejecutar el siguiente comando y se enviarán todos los *commits* que hayas hecho al servidor:

```
$ git push origin master
```

Este comando solo funciona si clonaste de un servidor sobre el que tienes permisos de escritura y si nadie más ha enviado datos por el medio. Si alguien más clona el mismo repositorio que tú y envía información antes que tú, tu envío será rechazado. Tendrás que traerte su trabajo y combinarlo con el tuyo antes de que puedas enviar datos al servidor. Para información más detallada sobre cómo enviar datos a servidores remotos, véase Ramificaciones en Git.

Inspeccionar un Remoto

Si quieres ver más información acerca de un remoto en particular, puedes ejecutar el comando `git remote show [nombre-remoto]`. Si ejecutas el comando con un nombre en particular, como `origin`, verás algo como lo siguiente:

```
$ git remote show origin
* remote origin
  Fetch URL: https://github.com/schacon/ticgit
  Push  URL: https://github.com/schacon/ticgit
  HEAD branch: master
  Remote branches:
    master                               tracked
    dev-branch                           tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

El comando lista la URL del repositorio remoto y la información del rastreo de ramas. El comando te indica claramente que si estás en la rama maestra y ejecutas el comando `git pull`, automáticamente combinará la rama maestra remota luego de haber traído toda la información de ella. También lista todas las referencias remotas de las que ha traído datos.

Ejemplos como este son los que te encontrarás normalmente. Sin embargo, si usas Git de forma más avanzada, puede que obtengas mucha más información de un `git remote show`:

```
$ git remote show origin
* remote origin
  URL: https://github.com/my-org/complex-project
  Fetch URL: https://github.com/my-org/complex-project
  Push  URL: https://github.com/my-org/complex-project
  HEAD branch: master
  Remote branches:
    master                           tracked
    dev-branch                       tracked
    markdown-strip                   tracked
    issue-43                         new (next fetch will store in remotes/origin)
    issue-45                         new (next fetch will store in remotes/origin)
    refs/remotes/origin/issue-11     stale (use 'git remote prune' to remove)
  Local branches configured for 'git pull':
    dev-branch merges with remote dev-branch
    master     merges with remote master
  Local refs configured for 'git push':
    dev-branch                     pushes to dev-branch                     (up to
date)
    markdown-strip                 pushes to markdown-strip                 (up to
date)
    master                         pushes to master                         (up to
date)
```

Este comando te indica a cuál rama enviarás información automáticamente cada vez que ejecutas `git push`, dependiendo de la rama en la que estés. También te muestra cuáles ramas remotas no tienes aun, cuáles ramas remotas tienes que han sido eliminadas del servidor, y varias ramas que serán combinadas automáticamente cuando ejecutes `git pull`.

Eliminar y Renombrar Remotos

Si quieres cambiar el nombre de la referencia de un remoto puedes ejecutar `git remote rename`. Por ejemplo, si quieres cambiar el nombre de `pb` a `paul`, puedes hacerlo con `git remote rename`:

```
$ git remote rename pb paul
$ git remote
origin
paul
```

Es importante destacar que al hacer esto también cambias el nombre de las ramas remotas. Por lo tanto, lo que antes estaba referenciado como `pb/master` ahora lo está como `paul/master`.

Si por alguna razón quieres eliminar un remoto - has cambiado de servidor o no quieres seguir utilizando un *mirror*, o quizás un colaborador a dejado de trabajar en el proyecto - puedes usar `git remote rm`:

```
$ git remote rm paul
$ git remote
origin
```

Etiquetado

Como muchos VCS, Git tiene la posibilidad de etiquetar puntos específicos del historial como importantes. Esta funcionalidad se usa típicamente para marcar versiones de lanzamiento (v1.0, por ejemplo). En esta sección, aprenderás cómo listar las etiquetas disponibles, cómo crear nuevas etiquetas y cuáles son los distintos tipos de etiquetas.

Listar Tus Etiquetas

Listar las etiquetas disponibles en Git es sencillo. Simplemente escribe `git tag`:

```
$ git tag
v0.1
v1.3
```

Este comando lista las etiquetas en orden alfabético; el orden en el que aparecen no tiene mayor importancia.

También puedes buscar etiquetas con un patrón particular. El repositorio del código fuente de Git, por ejemplo, contiene más de 500 etiquetas. Si solo te interesa ver la serie 1.8.5, puedes ejecutar:

```
$ git tag -l 'v1.8.5*'
v1.8.5
v1.8.5-rc0
v1.8.5-rc1
v1.8.5-rc2
v1.8.5-rc3
v1.8.5.1
v1.8.5.2
v1.8.5.3
v1.8.5.4
v1.8.5.5
```

Crear Etiquetas

Git utiliza dos tipos principales de etiquetas: ligeras y anotadas.

Una etiqueta ligera es muy parecido a una rama que no cambia - simplemente es un puntero a un *commit* específico.

Sin embargo, las etiquetas anotadas se guardan en la base de datos de Git como objetos enteros. Tienen un *checksum*; contienen el nombre del etiquetador, correo electrónico y fecha; tienen un

mensaje asociado; y pueden ser firmadas y verificadas con *GNU Privacy Guard* (GPG). Normalmente se recomienda que crees etiquetas anotadas, de manera que tengas toda esta información; pero si quieres una etiqueta temporal o por alguna razón no estás interesado en esa información, entonces puedes usar las etiquetas ligeras.

Etiquetas Anotadas

Crear una etiqueta anotada en Git es sencillo. La forma más fácil de hacer es especificar la opción `-a` cuando ejecutas el comando `tag`:

```
$ git tag -a v1.4 -m 'my version 1.4'
$ git tag
v0.1
v1.3
v1.4
```

La opción `-m` especifica el mensaje de la etiqueta, el cual es guardado junto con ella. Si no especificas el mensaje de una etiqueta anotada, Git abrirá el editor de texto para que lo escribas.

Puedes ver la información de la etiqueta junto con el *commit* que está etiquetado al usar el comando `git show`:

```
$ git show v1.4
tag v1.4
Tagger: Ben Straub <ben@straub.cc>
Date:   Sat May 3 20:19:12 2014 -0700

my version 1.4

commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number
```

El comando muestra la información del etiquetador, la fecha en la que el *commit* fue etiquetado y el mensaje de la etiquetar, antes de mostrar la información del *commit*.

Etiquetas Ligeras

La otra forma de etiquetar un *commit* es mediante una etiqueta ligera. Una etiqueta ligera no es más que el *checksum* de un *commit* guardado en un archivo - no incluye más información. Para crear una etiqueta ligera, no pases las opciones `-a`, `-s` ni `-m`:

```
$ git tag v1.4-lw
$ git tag
v0.1
v1.3
v1.4
v1.4-lw
v1.5
```

Esta vez, si ejecutas `git show` sobre la etiqueta, no verás la información adicional. El comando solo mostrará el *commit*:

```
$ git show v1.4-lw
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number
```

Etiquetado Tardío

También puedes etiquetar *commits* mucho tiempo después de haberlos hecho. Supongamos que tu historial luce como el siguiente:

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

Ahora, supongamos que olvidaste etiquetar el proyecto en su versión v1.2, la cual corresponde al *commit* “updated rakefile”. Igual puedes etiquetarlo. Para etiquetar un commit, debes especificar el *checksum* del *commit* (o parte de él) al final del comando:

```
$ git tag -a v1.2 9fceb02
```

Puedes ver que has etiquetado el commit:

```
$ git tag
v0.1
v1.2
v1.3
v1.4
v1.4-lw
v1.5

$ git show v1.2
tag v1.2
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 15:32:16 2009 -0800

version 1.2
commit 9fceb02d0ae598e95dc970b74767f19372d61af8
Author: Magnus Chacon <mchacon@gee-mail.com>
Date:   Sun Apr 27 20:43:35 2008 -0700

    updated rakefile
...
```

Compartir Etiquetas

Por defecto, el comando `git push` no transfiere las etiquetas a los servidores remotos. Debes enviar las etiquetas de forma explícita al servidor luego de que las hayas creado. Este proceso es similar al de compartir ramas remotas - puede ejecutar `git push origin [etiqueta]`.

```
$ git push origin v1.5
Counting objects: 14, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (12/12), done.
Writing objects: 100% (14/14), 2.05 KiB | 0 bytes/s, done.
Total 14 (delta 3), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new tag]         v1.5 -> v1.5
```

Si quieres enviar varias etiquetas a la vez, puedes usar la opción `--tags` del comando `git push`. Esto enviará al servidor remoto todas las etiquetas que aun no existen en él.

```
$ git push origin --tags
Counting objects: 1, done.
Writing objects: 100% (1/1), 160 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new tag]         v1.4 -> v1.4
 * [new tag]         v1.4-lw -> v1.4-lw
```

Por lo tanto, cuando alguien clone o traiga información de tu repositorio, también obtendrá todas las etiquetas.

Sacar una Etiqueta

En Git, no puedes sacar (*check out*) una etiqueta, pues no es algo que puedas mover. Si quieres colocar en tu directorio de trabajo una versión de tu repositorio que coincida con alguna etiqueta, debes crear una rama nueva en esa etiqueta:

```
$ git checkout -b version2 v2.0.0
Switched to a new branch 'version2'
```

Obviamente, si haces esto y luego confirmas tus cambios, tu rama `version2` será ligeramente distinta a tu etiqueta `v2.0.0` puesto que incluirá tus nuevos cambios; así que ten cuidado.

Alias de Git

Antes de terminar este capítulo sobre fundamentos de Git, hay otro pequeño consejo que puede hacer que tu experiencia con Git sea más simple, sencilla y familiar: los alias. No volveremos a mencionarlos más adelante en este libro, ni supondremos que los has utilizado, pero probablemente deberías saber cómo utilizarlos.

Git no deduce automáticamente tu comando si lo tecleas parcialmente. Si no quieres teclear el nombre completo de cada comando de Git, puedes establecer fácilmente un alias para cada comando mediante `git config`. Aquí tienes algunos ejemplos que te pueden interesar:

```
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.ci commit
$ git config --global alias.st status
```

Esto significa que, por ejemplo, en lugar de teclear `git commit`, solo necesitas teclear `git ci`. A medida que uses Git, probablemente también utilizarás otros comandos con frecuencia; no dudes en crear nuevos alias.

Esta técnica también puede resultar útil para crear comandos que en tu opinión deberían existir. Por ejemplo, para corregir el problema de usabilidad que encontraste al quitar del área de preparación un archivo, puedes añadir tu propio alias a Git:

```
$ git config --global alias.unstage 'reset HEAD --'
```

Esto hace que los dos comandos siguientes sean equivalentes:

```
$ git unstage fileA
$ git reset HEAD fileA
```

Esto parece un poco más claro. También es frecuente añadir un comando `last`, de este modo:

```
$ git config --global alias.last 'log -1 HEAD'
```

De esta manera, puedes ver fácilmente cuál fue la última confirmación:

```
$ git last
commit 66938dae3329c7aebe598c2246a8e6af90d04646
Author: Josh Goebel <dreamer3@example.com>
Date:   Tue Aug 26 19:48:51 2008 +0800

    test for current head

    Signed-off-by: Scott Chacon <schacon@example.com>
```

Como puedes ver, Git simplemente sustituye el nuevo comando por lo que sea que hayas puesto en el alias. Sin embargo, quizás quieras ejecutar un comando externo, en lugar de un subcomando de Git. En ese caso, puedes comenzar el comando con un carácter `!`. Esto resulta útil si escribes tus propias herramientas para trabajar con un repositorio de Git. Podemos demostrarlo creando el alias `git visual` para ejecutar `gitk`:

```
$ git config --global alias.visual "!gitk"
```

Resumen

En este momento puedes hacer todas las operaciones básicas de Git a nivel local: Crear o clonar un repositorio, hacer cambios, preparar y confirmar esos cambios y ver la historia de los cambios en el repositorio. A continuación cubriremos la mejor característica de Git: Su modelo de ramas.

Ramificaciones en Git

Cualquier sistema de control de versiones moderno tiene algún mecanismo para soportar el uso de ramas. Cuando hablamos de ramificaciones, significa que tú has tomado la rama principal de desarrollo (master) y a partir de ahí has continuado trabajando sin seguir la rama principal de desarrollo. En muchos sistemas de control de versiones este proceso es costoso, pues a menudo requiere crear una nueva copia del código, lo cual puede tomar mucho tiempo cuando se trata de proyectos grandes.

Algunas personas resaltan que uno de los puntos más fuertes de Git es su sistema de ramificaciones y lo cierto es que esto le hace resaltar sobre los otros sistemas de control de versiones. ¿Por qué esto es tan importante? La forma en la que Git maneja las ramificaciones es increíblemente rápida, haciendo así de las operaciones de ramificación algo casi instantáneo, al igual que el avance o el retroceso entre distintas ramas, lo cual también es tremendamente rápido. A diferencia de otros sistemas de control de versiones, Git promueve un ciclo de desarrollo donde las ramas se crean y se unen ramas entre sí, incluso varias veces en el mismo día. Entender y manejar esta opción te proporciona una poderosa y exclusiva herramienta que puede, literalmente, cambiar la forma en la que desarrollas.

¿Qué es una rama?

Para entender realmente cómo ramifica Git, previamente hemos de examinar la forma en que almacena sus datos.

Recordando lo citado en Inicio - Sobre el Control de Versiones, Git no los almacena de forma incremental (guardando solo diferencias), sino que los almacena como una serie de instantáneas (copias puntuales de los archivos completos, tal y como se encuentran en ese momento).

En cada confirmación de cambios (commit), Git almacena una instantánea de tu trabajo preparado. Dicha instantánea contiene además unos metadatos con el autor y el mensaje explicativo, y uno o varios apuntadores a las confirmaciones (commit) que sean padres directos de esta (un padre en los casos de confirmación normal, y múltiples padres en los casos de estar confirmando una fusión (merge) de dos o más ramas).

Para ilustrar esto, vamos a suponer, por ejemplo, que tienes una carpeta con tres archivos, que preparas (stage) todos ellos y los confirmas (commit). Al preparar los archivos, Git realiza una suma de control de cada uno de ellos (un resumen SHA-1, tal y como se mencionaba en Inicio - Sobre el Control de Versiones), almacena una copia de cada uno en el repositorio (estas copias se denominan "blobs"), y guarda cada suma de control en el área de preparación (staging area):

```
$ git add README test.rb LICENSE
$ git commit -m 'initial commit of my project'
```

Cuando creas una confirmación con el comando `git commit`, Git realiza sumas de control de cada subdirectorio (en el ejemplo, solamente tenemos el directorio principal del proyecto), y las guarda como objetos árbol en el repositorio Git. Después, Git crea un objeto de confirmación con los metadatos pertinentes y un apuntador al objeto árbol raíz del proyecto.

En este momento, el repositorio de Git contendrá cinco objetos: un "blob" para cada uno de los tres archivos, un árbol con la lista de contenidos del directorio (más sus respectivas relaciones con los "blobs"), y una confirmación de cambios (commit) apuntando a la raíz de ese árbol y conteniendo el resto de metadatos pertinentes.

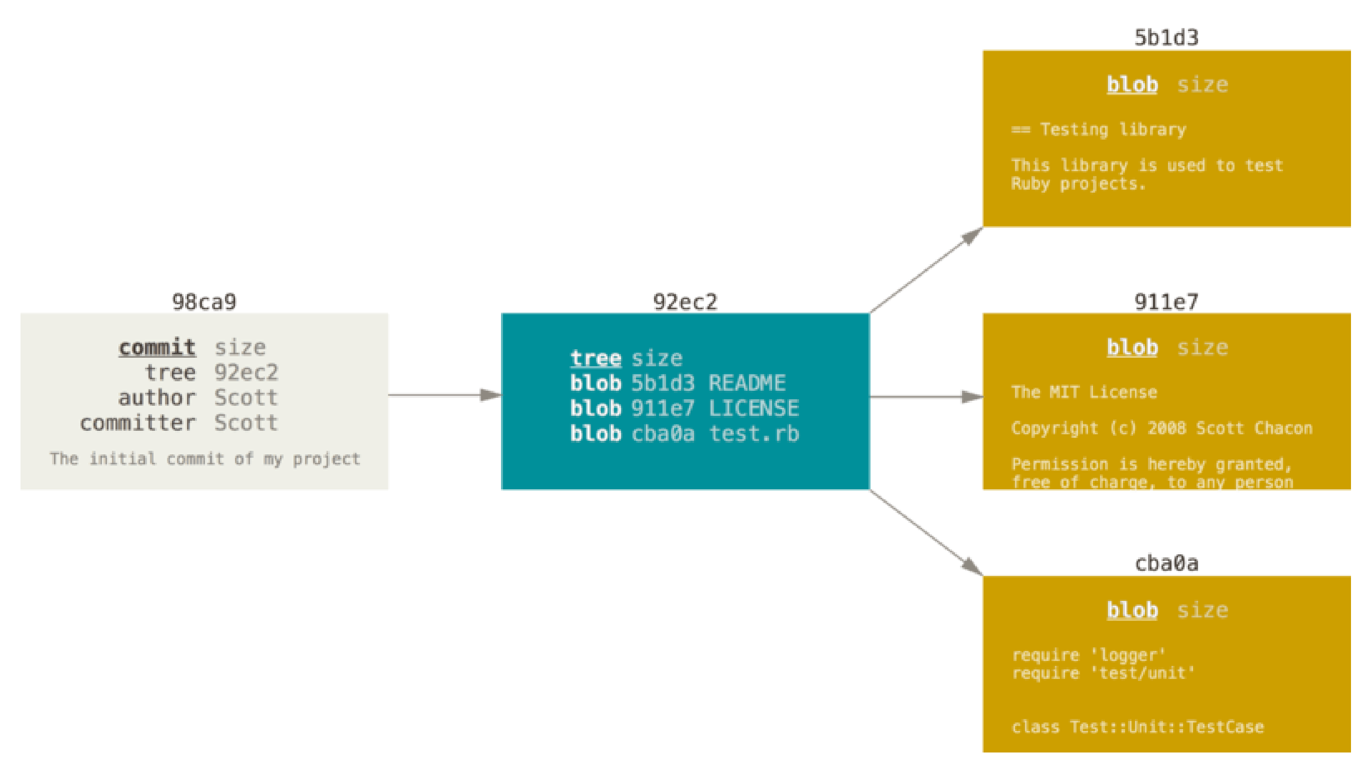


Figure 9. Una confirmación y sus árboles

Si haces más cambios y vuelves a confirmar, la siguiente confirmación guardará un apuntador su confirmación precedente.

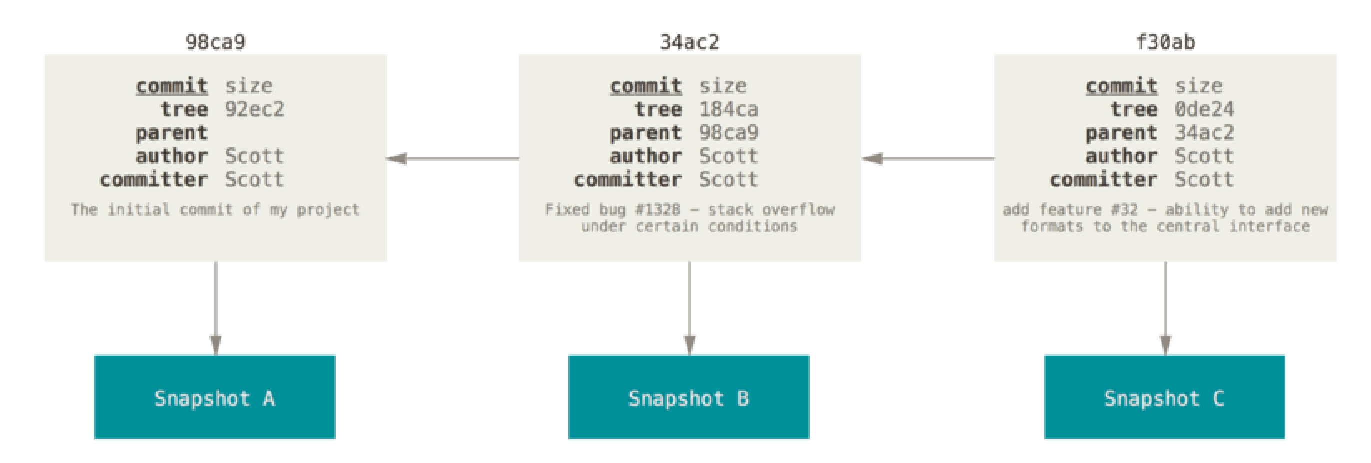


Figure 10. Confirmaciones y sus predecesoras

Una rama Git es simplemente un apuntador móvil apuntando a una de esas confirmaciones. La rama por defecto de Git es la rama `master`. Con la primera confirmación de cambios que realicemos, se creará esta rama principal `master` apuntando a dicha confirmación. En cada confirmación de cambios que realicemos, la rama irá avanzando automáticamente.

NOTE

La rama “master” en Git no es una rama especial. Es como cualquier otra rama. La única razón por la cual aparece en casi todos los repositorios es porque es la que crea por defecto el comando `git init` y la gente no se molesta en cambiarle el nombre.

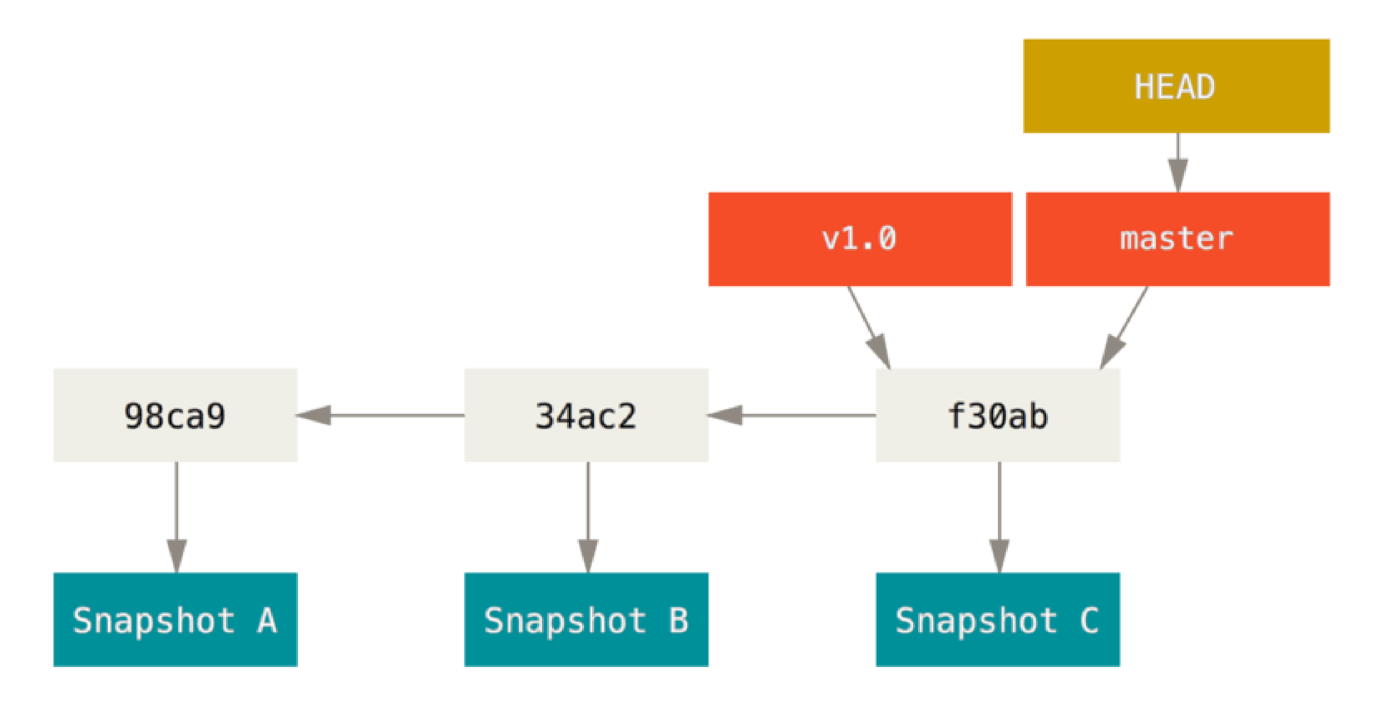


Figure 11. Una rama y su historial de confirmaciones

Crear una Rama Nueva

¿Qué sucede cuando creas una nueva rama? Bueno..., simplemente se crea un nuevo apuntador para que lo puedas mover libremente. Por ejemplo, supongamos que quieres crear una rama nueva denominada "testing". Para ello, usarás el comando `git branch`:

```
$ git branch testing
```

Esto creará un nuevo apuntador apuntando a la misma confirmación donde estés actualmente.

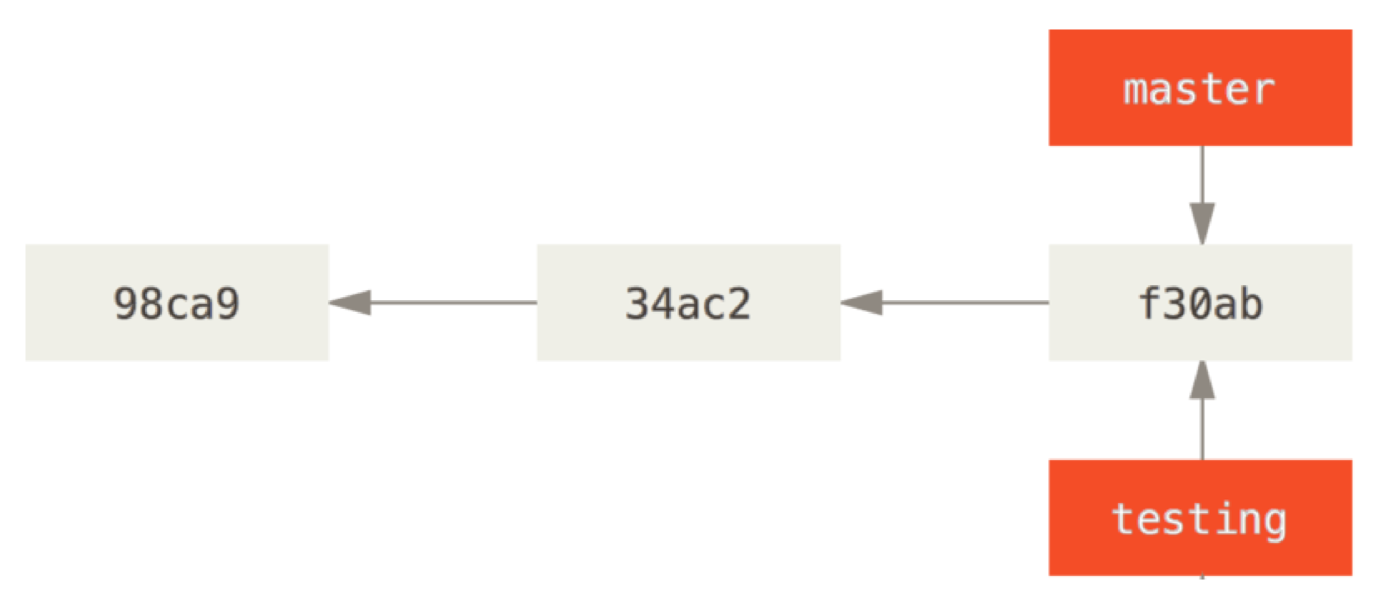


Figure 12. Dos ramas apuntando al mismo grupo de confirmaciones

Y, ¿cómo sabe Git en qué rama estás en este momento? Pues..., mediante un apuntador especial denominado HEAD. Aunque es preciso comentar que este HEAD es totalmente distinto al concepto de HEAD en otros sistemas de control de cambios como Subversion o CVS. En Git, es simplemente el apuntador a la rama local en la que tú estés en ese momento, en este caso la rama `master`; pues el comando `git branch` solamente crea una nueva rama, y no salta a dicha rama.

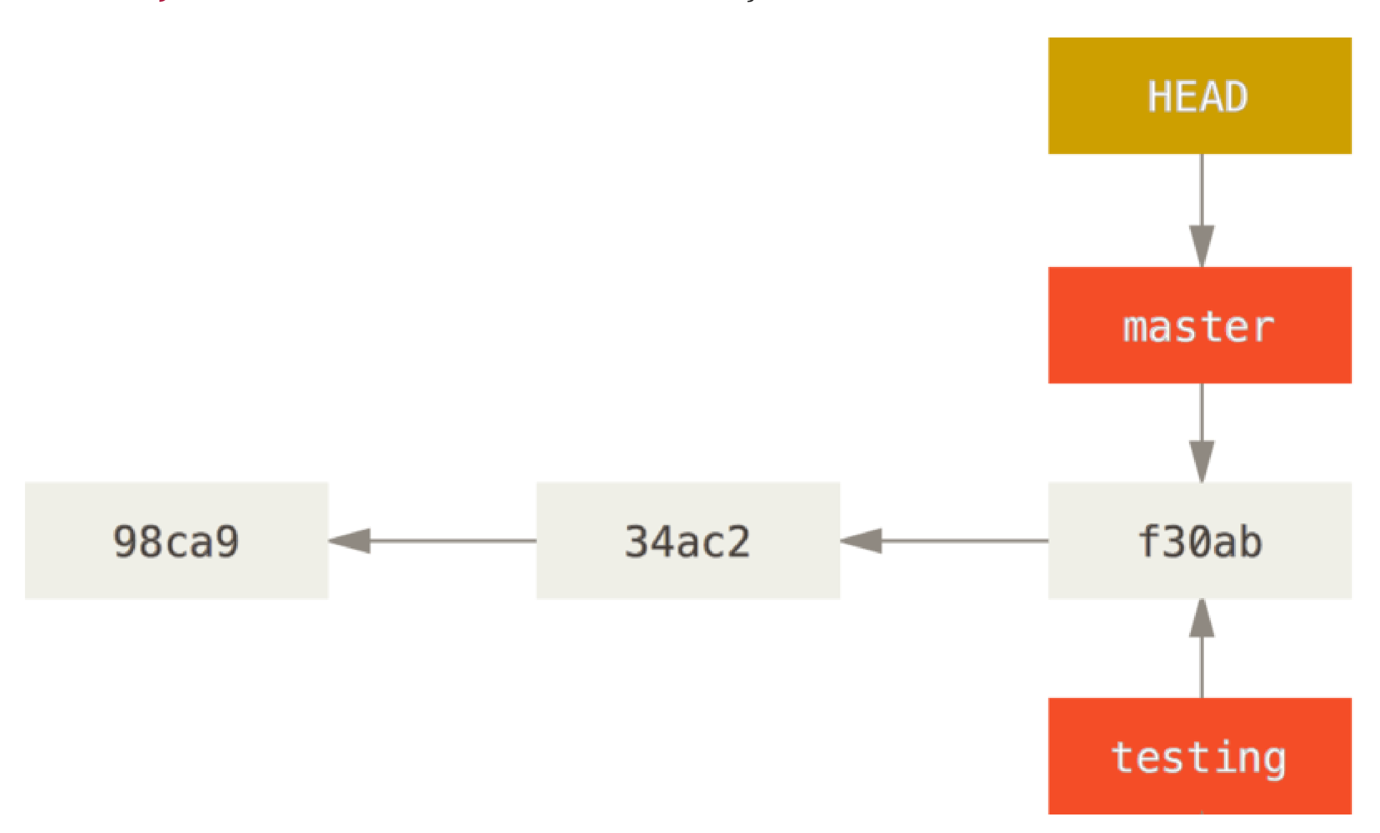


Figure 13. Apuntador HEAD a la rama donde estás actualmente

Esto puedes verlo fácilmente al ejecutar el comando `git log` para que te muestre a dónde apunta cada rama. Esta opción se llama `--decorate`.

```
$ git log --oneline --decorate
f30ab (HEAD, master, testing) add feature #32 - ability to add new
34ac2 fixed bug #1328 - stack overflow under certain conditions
98ca9 initial commit of my project
```

Puedes ver que las ramas “master” y “testing” están junto a la confirmación `f30ab`.

Cambiar de Rama

Para saltar de una rama a otra, tienes que utilizar el comando `git checkout`. Hagamos una prueba, saltando a la rama `testing` recién creada:

```
$ git checkout testing
```

Esto mueve el apuntador HEAD a la rama `testing`.

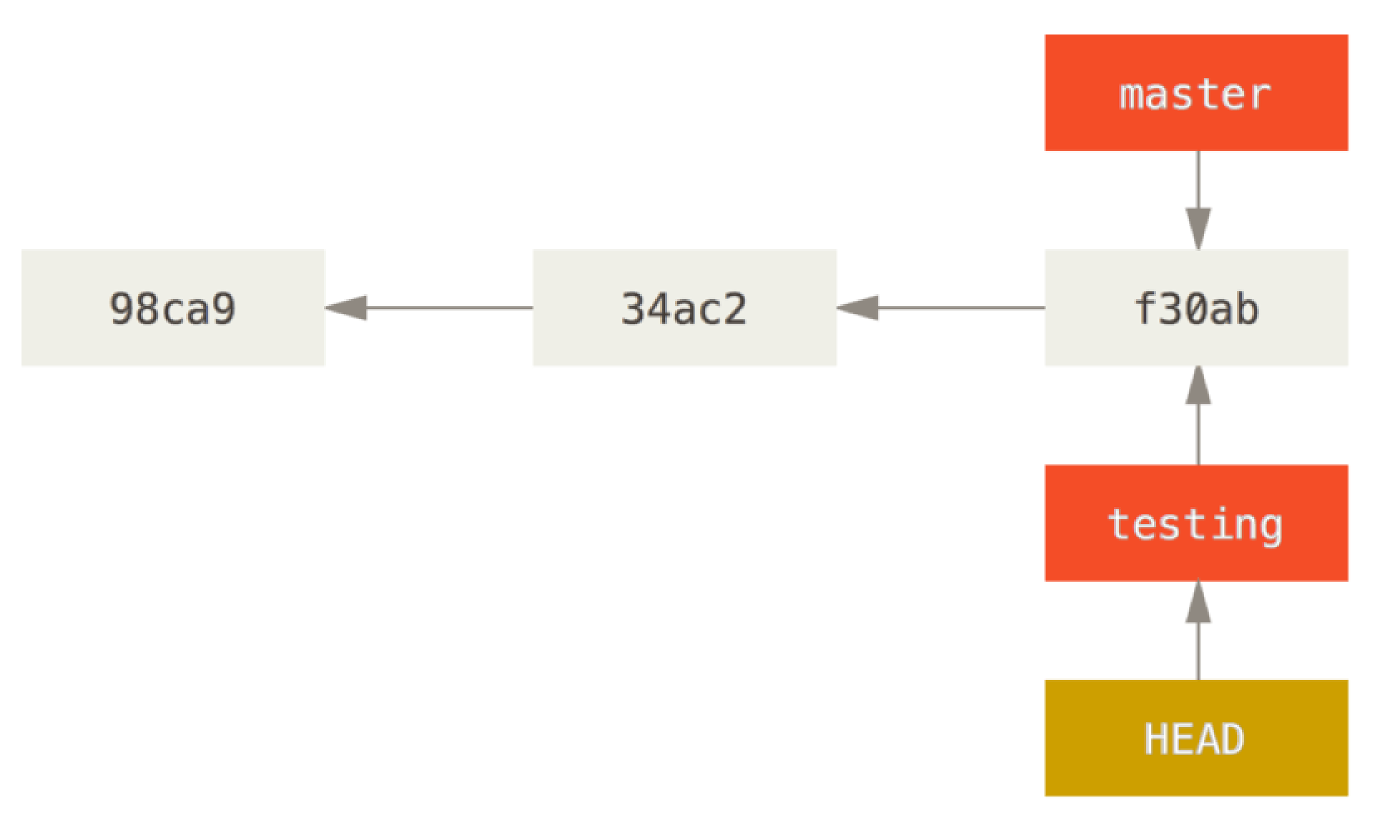


Figure 14. El apuntador HEAD apunta a la rama actual

¿Cuál es el significado de todo esto? Bueno… lo veremos tras realizar otra confirmación de cambios:

```
$ vim test.rb
$ git commit -a -m 'made a change'
```

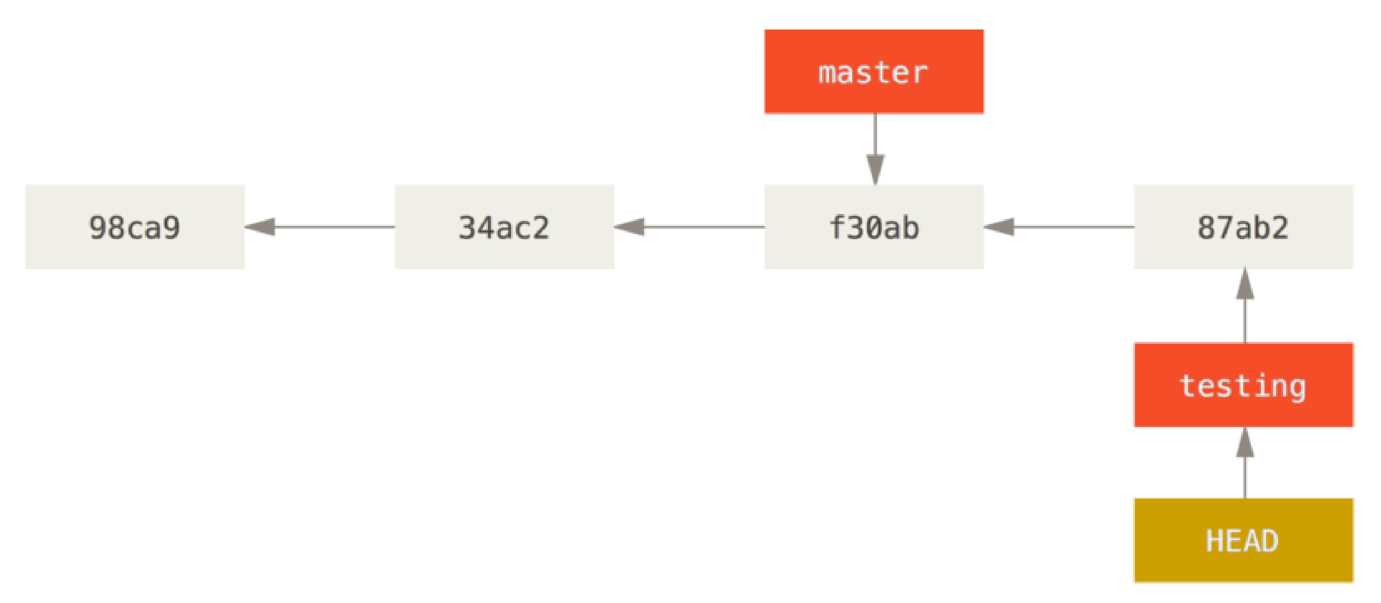


Figure 15. La rama apuntada por HEAD avanza con cada confirmación de cambios

Observamos algo interesante: la rama `testing` avanza, mientras que la rama `master` permanece en la confirmación donde estaba cuando lanzaste el comando `git checkout` para saltar. Volvamos ahora a la rama `master`:

```
$ git checkout master
```

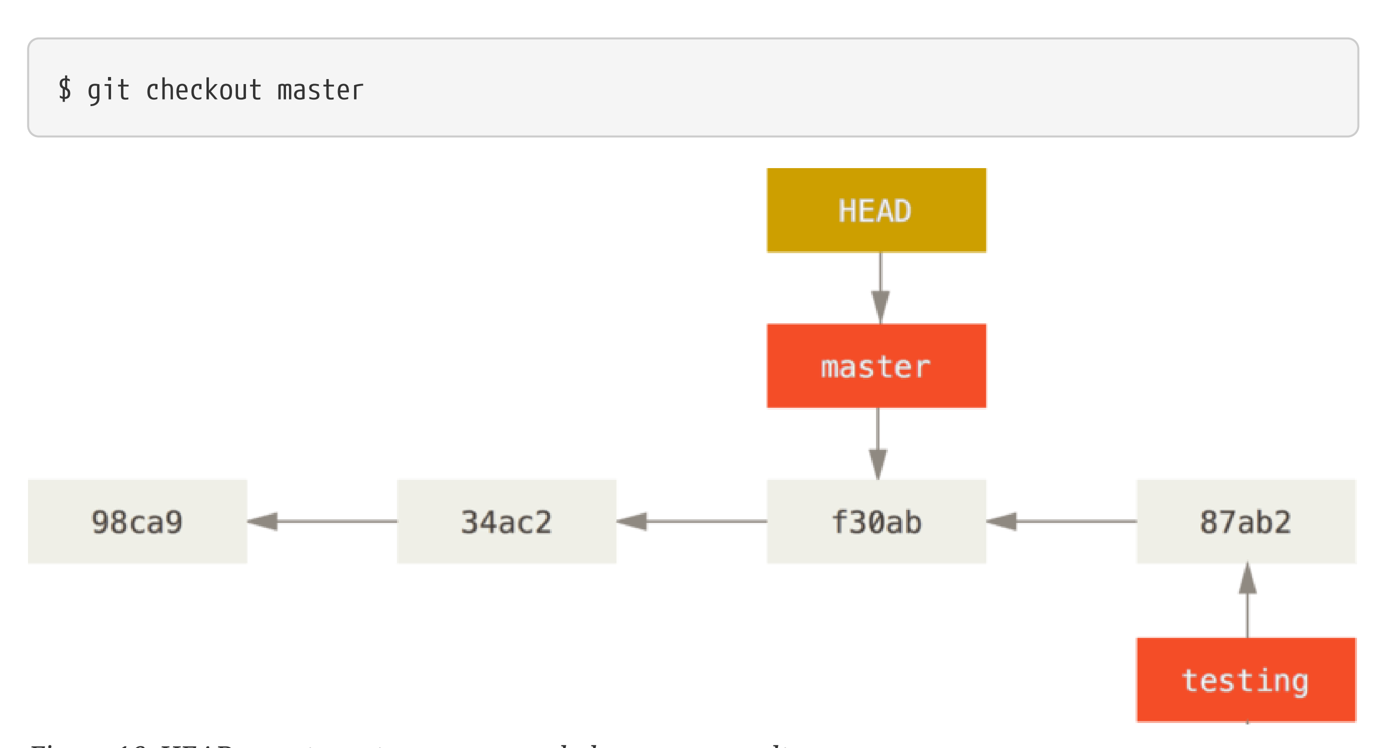


Figure 16. HEAD apunta a otra rama cuando hacemos un salto

Este comando realiza dos acciones: Mueve el apuntador HEAD de nuevo a la rama `master`, y revierte los archivos de tu directorio de trabajo; dejándolos tal y como estaban en la última instantánea confirmada en dicha rama `master`. Esto supone que los cambios que hagas desde este momento en adelante divergirán de la antigua versión del proyecto. Básicamente, lo que se está haciendo es rebobinar el trabajo que habías hecho temporalmente en la rama `testing`; de tal forma que puedas avanzar en otra dirección diferente.

NOTE	*Saltar entre ramas cambia archivos en tu directorio de trabajo* Es importante destacar que cuando saltas a una rama en Git, los archivos de tu directorio de trabajo cambian. Si saltas a una rama antigua, tu directorio de trabajo retrocederá para verse como lo hacía la última vez que confirmaste un cambio en dicha rama. Si Git no puede hacer el cambio limpiamente, no te dejará saltar.

Haz algunos cambios más y confírmalos:

```
$ vim test.rb
$ git commit -a -m 'made other changes'
```

Ahora el historial de tu proyecto diverge (ver Los registros de las ramas divergen). Has creado una rama y saltado a ella, has trabajado sobre ella; has vuelto a la rama original, y has trabajado también sobre ella. Los cambios realizados en ambas sesiones de trabajo están aislados en ramas independientes: puedes saltar libremente de una a otra según estimes oportuno. Y todo ello simplemente con tres comandos: `git branch`, `git checkout` y `git commit`.

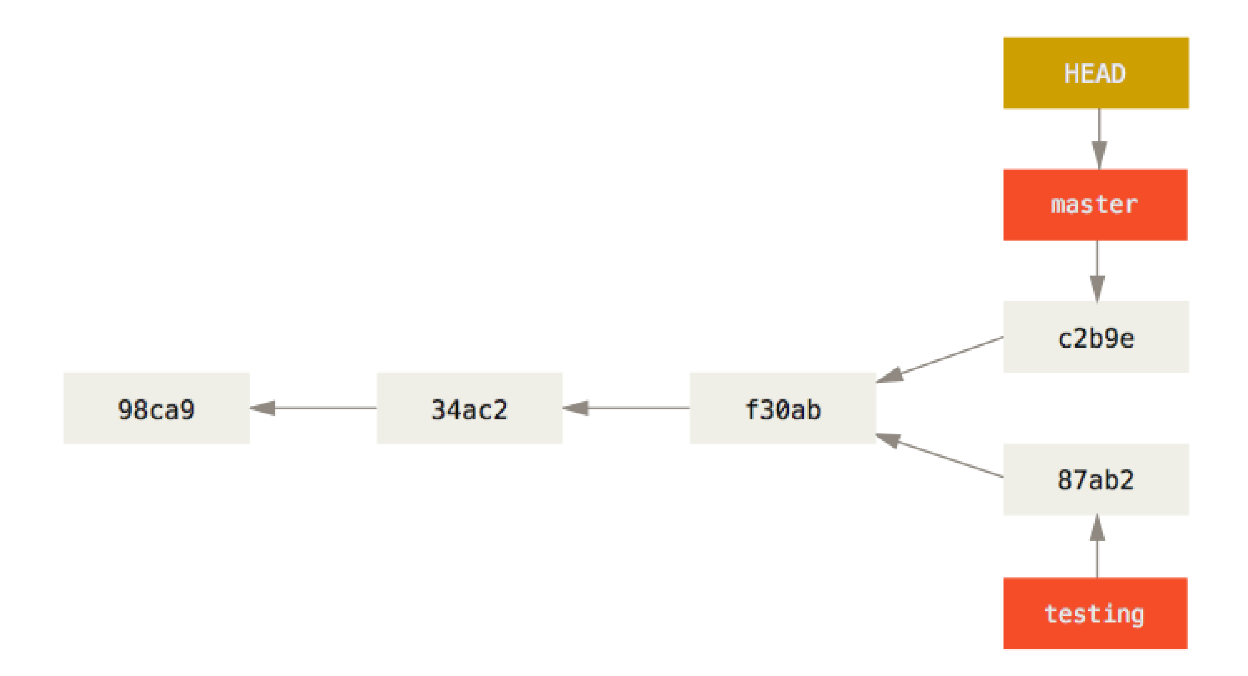


Figure 17. Los registros de las ramas divergen

También puedes ver esto fácilmente utilizando el comando `git log`. Si ejecutas `git log --oneline --decorate --graph --all` te mostrará el historial de tus confirmaciones, indicando dónde están los apuntadores de tus ramas y como ha divergido tu historial.

```
$ git log --oneline --decorate --graph --all
* c2b9e (HEAD, master) made other changes
| * 87ab2 (testing) made a change
|/
* f30ab add feature #32 - ability to add new formats to the
* 34ac2 fixed bug #1328 - stack overflow under certain conditions
* 98ca9 initial commit of my project
```

Debido a que una rama Git es realmente un simple archivo que contiene los 40 caracteres de una suma de control SHA-1, (representando la confirmación de cambios a la que apunta), no cuesta nada el crear y destruir ramas en Git. Crear una nueva rama es tan rápido y simple como escribir 41 bytes en un archivo, (40 caracteres y un retorno de carro).

Esto contrasta fuertemente con los métodos de ramificación usados por otros sistemas de control de versiones, en los que crear una rama nueva supone el copiar todos los archivos del proyecto a un directorio adicional nuevo. Esto puede llevar segundos o incluso minutos, dependiendo del tamaño del proyecto; mientras que en Git el proceso es siempre instantáneo. Y, además, debido a que se almacenan también los nodos padre para cada confirmación, el encontrar las bases adecuadas para realizar una fusión entre ramas es un proceso automático y generalmente sencillo de realizar. Animando así a los desarrolladores a utilizar ramificaciones frecuentemente.

Vamos a ver el por qué merece la pena hacerlo así.

Procedimientos Básicos para Ramificar y Fusionar

Vamos a presentar un ejemplo simple de ramificar y de fusionar, con un flujo de trabajo que se podría presentar en la realidad. Imagina que sigues los siguientes pasos:

1. Trabajas en un sitio web.
2. Creas una rama para un nuevo tema sobre el que quieres trabajar.
3. Realizas algo de trabajo en esa rama.

En este momento, recibes una llamada avisándote de un problema crítico que has de resolver. Y sigues los siguientes pasos:

1. Vuelves a la rama de producción original.
2. Creas una nueva rama para el problema crítico y lo resuelves trabajando en ella.
3. Tras las pertinentes pruebas, fusionas (merge) esa rama y la envías (push) a la rama de producción.
4. Vuelves a la rama del tema en que andabas antes de la llamada y continuas tu trabajo.

Procedimientos Básicos de Ramificación

Imagina que estas trabajando en un proyecto y tienes un par de confirmaciones (commit) ya realizadas.

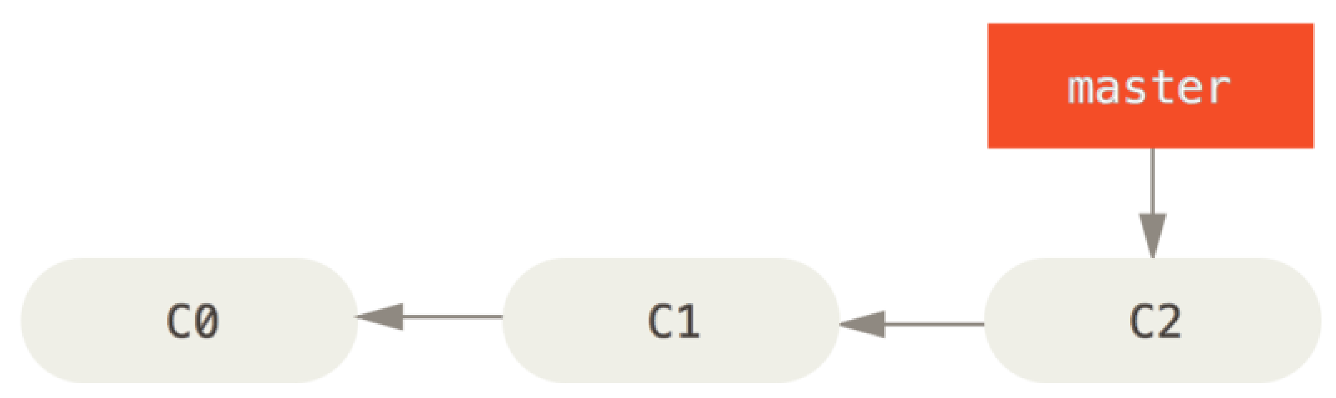


Figure 18. Un registro de confirmaciones corto y sencillo

Decides trabajar en el problema #53, según el sistema que tu compañía utiliza para llevar seguimiento de los problemas. Para crear una nueva rama y saltar a ella, en un solo paso, puedes utilizar el comando `git checkout` con la opción `-b`:

```
$ git checkout -b iss53
Switched to a new branch "iss53"
```

Esto es un atajo a:

```
$ git branch iss53
$ git checkout iss53
```

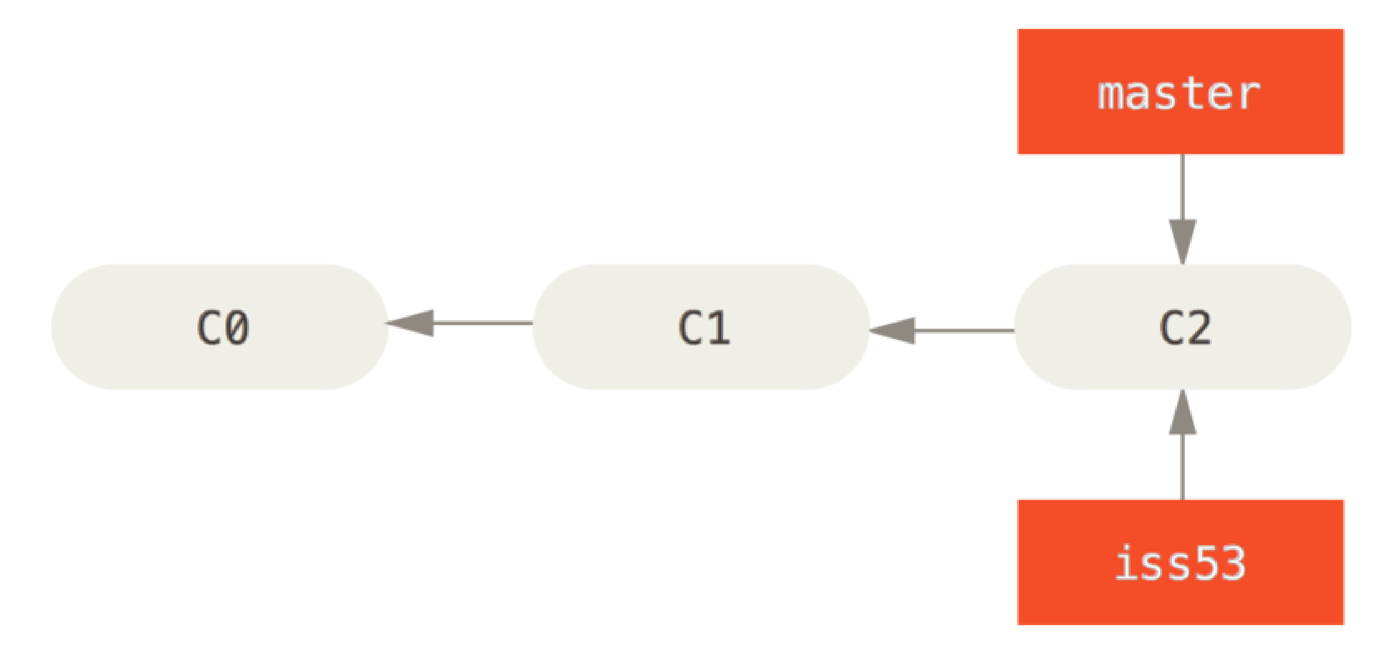


Figure 19. Crear un apuntador a la rama nueva

Trabajas en el sitio web y haces algunas confirmaciones de cambios (commits). Con ello avanzas la rama `iss53`, que es la que tienes activada (checked out) en este momento (es decir, a la que apunta HEAD):

```
$ vim index.html
$ git commit -a -m 'added a new footer [issue 53]'
```

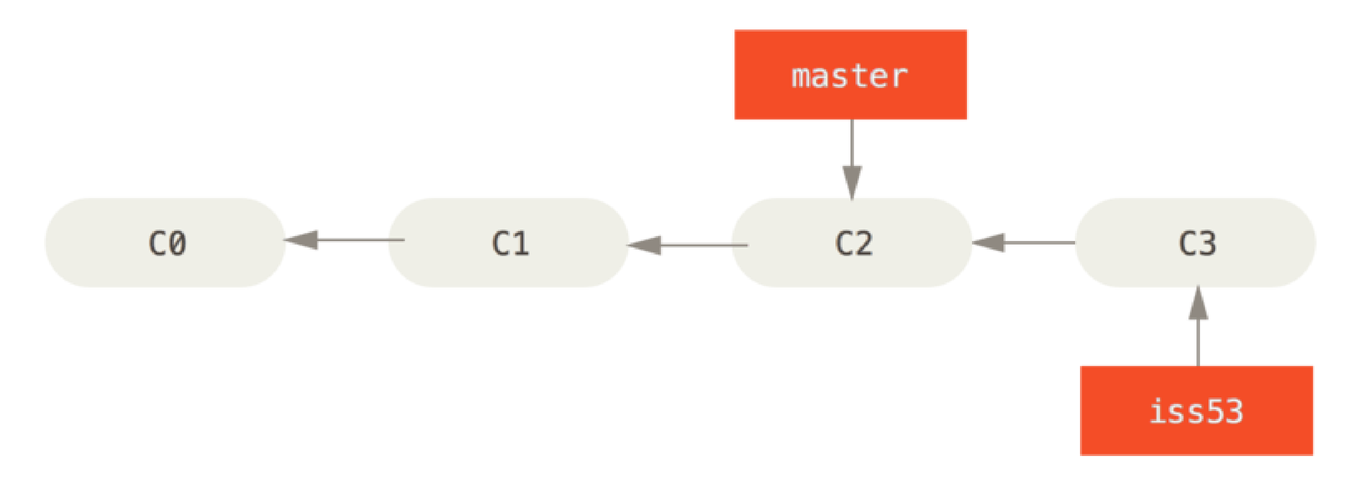


Figure 20. La rama `iss53` ha avanzado con tu trabajo

Entonces, recibes una llamada avisándote de otro problema urgente en el sitio web y debes resolverlo inmediatamente. Al usar Git, no necesitas mezclar el nuevo problema con los cambios que ya habías realizado sobre el problema #53; ni tampoco perder tiempo revirtiendo esos cambios para poder trabajar sobre el contenido que está en producción. Basta con saltar de nuevo a la rama `master` y continuar trabajando a partir de allí.

Pero, antes de poder hacer eso, hemos de tener en cuenta que si tenemos cambios aún no confirmados en el directorio de trabajo o en el área de preparación, Git no nos permitirá saltar a otra rama con la que podríamos tener conflictos. Lo mejor es tener siempre un estado de trabajo limpio y despejado antes de saltar entre ramas. Y, para ello, tenemos algunos procedimientos (stash y corregir confirmaciones), que vamos a ver más adelante en Guardado rápido y Limpieza. Por ahora, como tenemos confirmados todos los cambios, podemos saltar a la rama `master` sin problemas:

```
$ git checkout master
Switched to branch 'master'
```

Tras esto, tendrás el directorio de trabajo exactamente igual a como estaba antes de comenzar a trabajar sobre el problema #53 y podrás concentrarte en el nuevo problema urgente. Es importante recordar que Git revierte el directorio de trabajo exactamente al estado en que estaba en la confirmación (commit) apuntada por la rama que activamos (checkout) en cada momento. Git añade, quita y modifica archivos automáticamente para asegurar que tu copia de trabajo luce exactamente como lucía la rama en la última confirmación de cambios realizada sobre ella.

A continuación, es momento de resolver el problema urgente. Vamos a crear una nueva rama `hotfix`, sobre la que trabajar hasta resolverlo:

```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'fixed the broken email address'
[hotfix 1fb7853] fixed the broken email address
 1 file changed, 2 insertions(+)
```

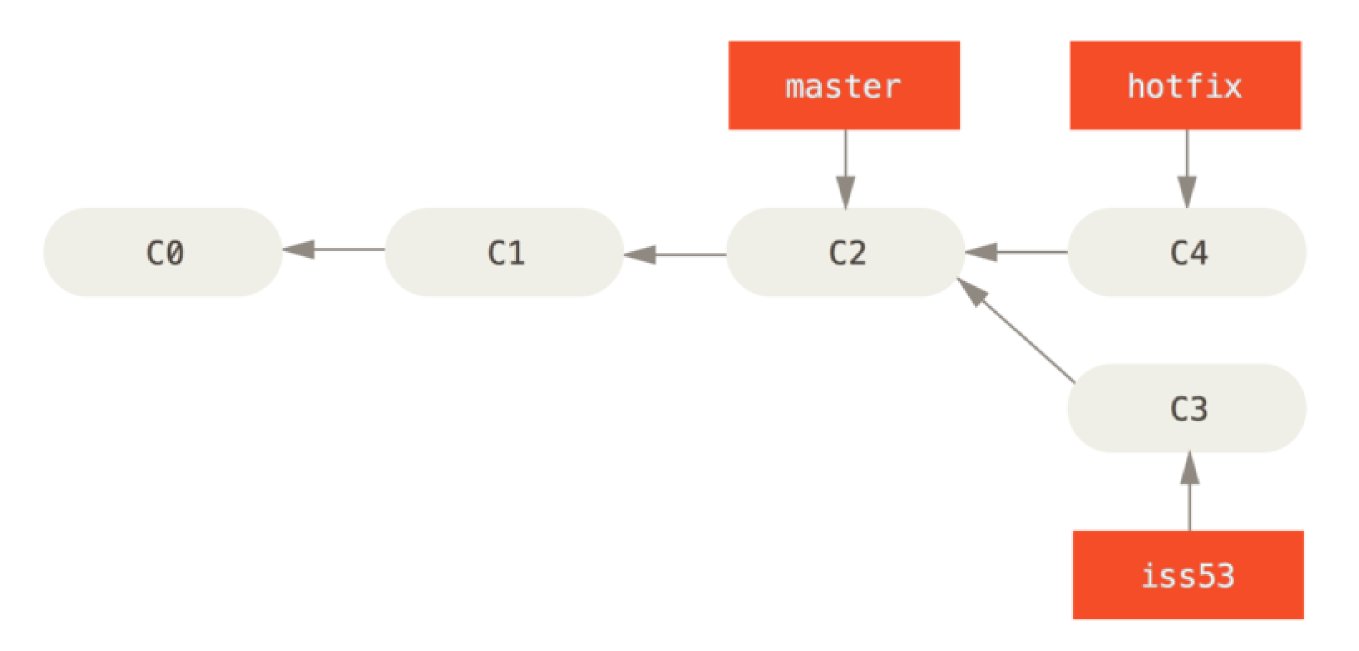


Figure 21. Rama `hotfix` basada en la rama `master` original

Puedes realizar las pruebas oportunas, asegurarte que la solución es correcta, e incorporar los cambios a la rama `master` para ponerlos en producción. Esto se hace con el comando `git merge`:

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 index.html | 2 ++
 1 file changed, 2 insertions(+)
```

Notarás la frase "Fast forward" ("Avance rápido", en inglés) que aparece en la salida del comando. Git ha movido el apuntador hacia adelante, ya que la confirmación apuntada en la rama donde has fusionado estaba directamente arriba respecto a la confirmación actual. Dicho de otro modo: cuando intentas fusionar una confirmación con otra confirmación accesible siguiendo directamente el historial de la primera; Git simplifica las cosas avanzando el puntero, ya que no hay ningún otro trabajo divergente a fusionar. Esto es lo que se denomina "avance rápido" ("fast forward").

Ahora, los cambios realizados están ya en la instantánea (snapshot) de la confirmación (commit) apuntada por la rama `master`. Y puedes desplegarlos.

Figure 22. Tras la fusión (merge), la rama `master` apunta al mismo sitio que la rama `hotfix`.

Tras haber resuelto el problema urgente que había interrumpido tu trabajo, puedes volver a donde estabas. Pero antes, es importante borrar la rama `hotfix`, ya que no la vamos a necesitar más, puesto que apunta exactamente al mismo sitio que la rama `master`. Esto lo puedes hacer con la opción `-d` del comando `git branch`:

```
$ git branch -d hotfix
Deleted branch hotfix (3a0874c).
```

Y, con esto, ya estás listo para regresar al trabajo sobre el problema #53.

```
$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'finished the new footer [issue 53]'
[iss53 ad82d7a] finished the new footer [issue 53]
1 file changed, 1 insertion(+)
```

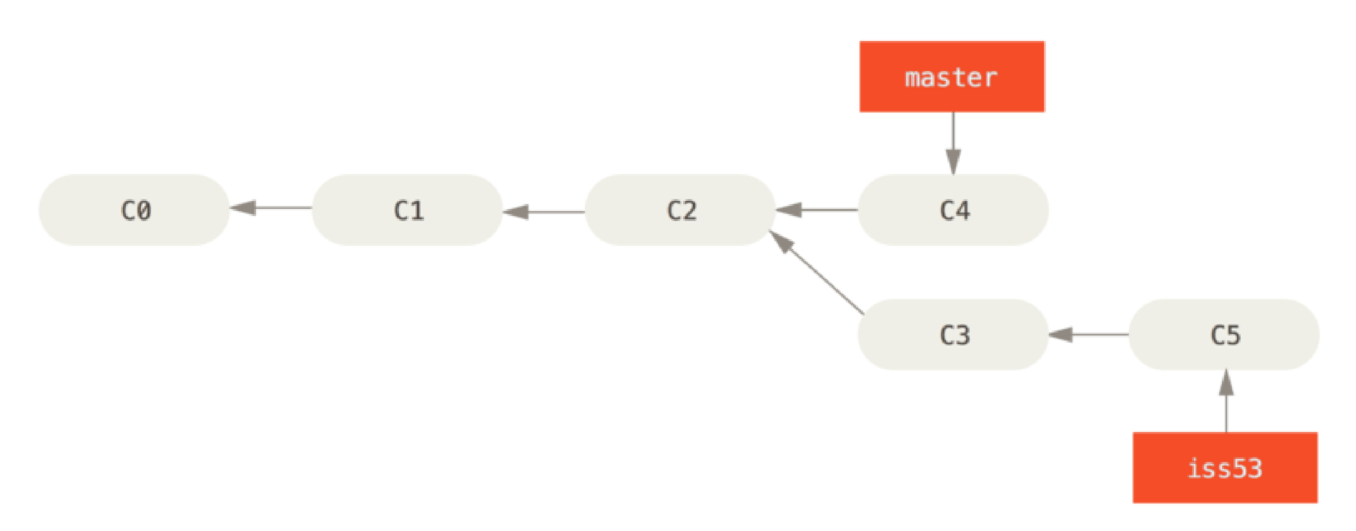


Figure 23. La rama `iss53` puede avanzar independientemente

Cabe destacar que todo el trabajo realizado en la rama `hotfix` no está en los archivos de la rama `iss53`. Si fuera necesario agregarlos, puedes fusionar (merge) la rama `master` sobre la rama `iss53` utilizando el comando `git merge master`, o puedes esperar hasta que decidas fusionar (merge) la rama `iss53` a la rama `master`.

Procedimientos Básicos de Fusión

Supongamos que tu trabajo con el problema #53 ya está completo y listo para fusionarlo (merge) con la rama `master`. Para ello, de forma similar a como antes has hecho con la rama `hotfix`, vas a fusionar la rama `iss53`. Simplemente, activa (checkout) la rama donde deseas fusionar y lanza el comando `git merge`:

```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
index.html |    1 +
1 file changed, 1 insertion(+)
```

Es algo diferente de la fusión realizada anteriormente con `hotfix`. En este caso, el registro de desarrollo había divergido en un punto anterior. Debido a que la confirmación en la rama actual no es ancestro directo de la rama que pretendes fusionar, Git tiene cierto trabajo extra que hacer. Git realizará una fusión a tres bandas, utilizando las dos instantáneas apuntadas por el extremo de cada una de las ramas y por el ancestro común a ambas.

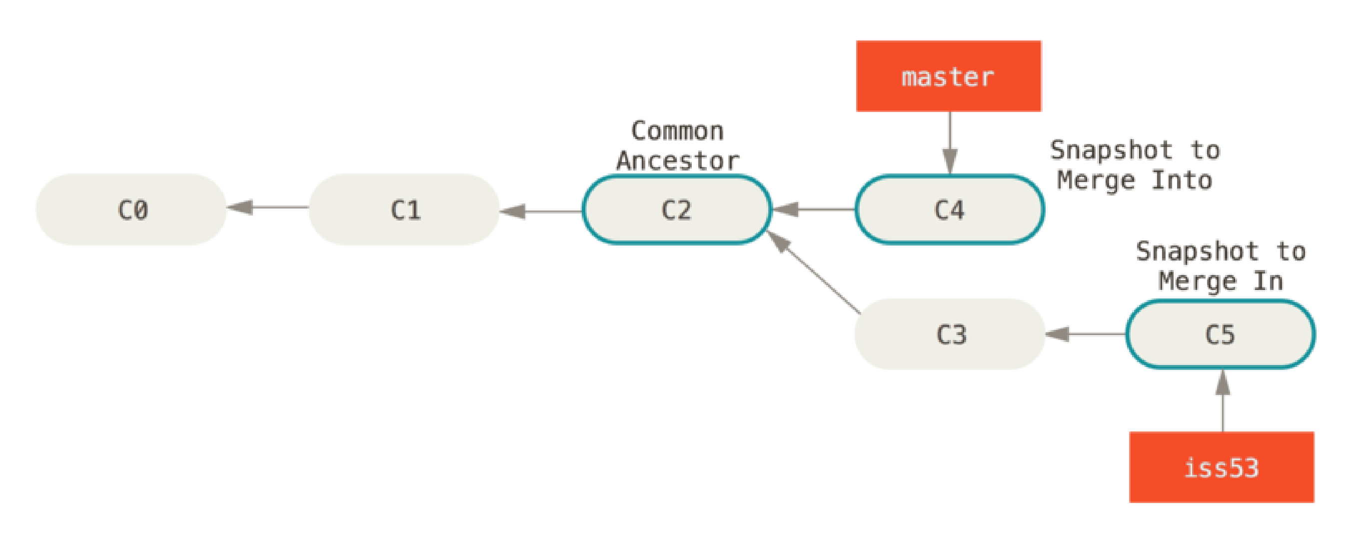


Figure 24. Git identifica automáticamente el mejor ancestro común para realizar la fusión de las ramas

En lugar de simplemente avanzar el apuntador de la rama, Git crea una nueva instantánea (snapshot) resultante de la fusión a tres bandas; y crea automáticamente una nueva confirmación de cambios (commit) que apunta a ella. Nos referimos a este proceso como "fusión confirmada" y su particularidad es que tiene más de un padre.

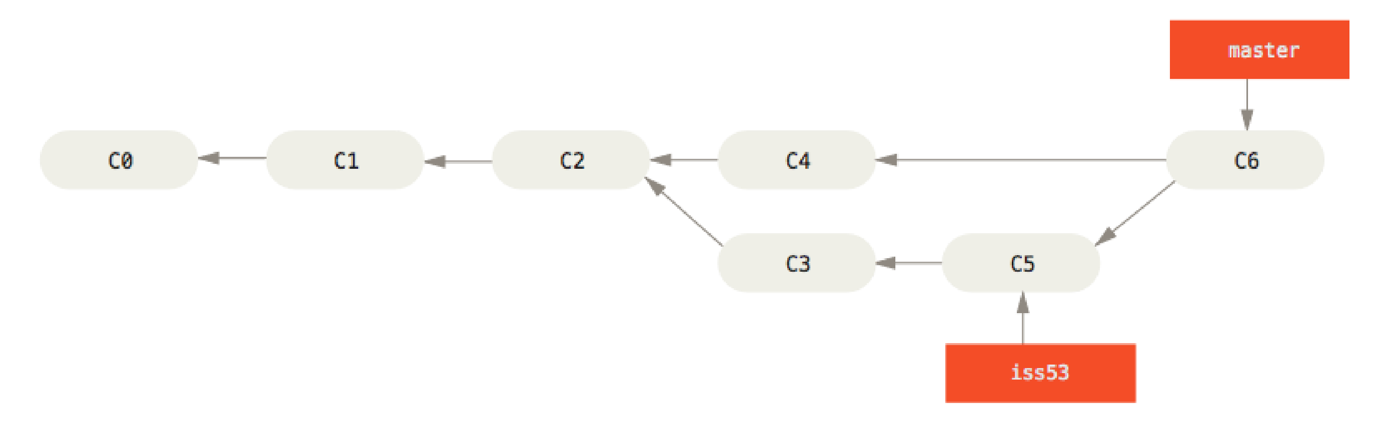


Figure 25. Git crea automáticamente una nueva confirmación para la fusión

Vale la pena destacar el hecho de que es el propio Git quien determina automáticamente el mejor ancestro común para realizar la fusión; a diferencia de otros sistemas tales como CVS o Subversion, donde es el desarrollador quien ha de determinar cuál puede ser dicho mejor ancestro común. Esto hace que en Git sea mucho más fácil realizar fusiones.

Ahora que todo tu trabajo ya está fusionado con la rama principal, no tienes necesidad de la rama `iss53`. Por lo que puedes borrarla y cerrar manualmente el problema en el sistema de seguimiento de problemas de tu empresa.

```
$ git branch -d iss53
```

Principales Conflictos que Pueden Surgir en las Fusiones

En algunas ocasiones, los procesos de fusión no suelen ser fluidos. Si hay modificaciones dispares en una misma porción de un mismo archivo en las dos ramas distintas que pretendes fusionar, Git no será capaz de fusionarlas directamente. Por ejemplo, si en tu trabajo del problema #53 has modificado una misma porción que también ha sido modificada en el problema `hotfix`, verás un conflicto como este:

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Git no crea automáticamente una nueva fusión confirmada (merge commit), sino que hace una pausa en el proceso, esperando a que tú resuelvas el conflicto. Para ver qué archivos permanecen sin fusionar en un determinado momento conflictivo de una fusión, puedes usar el comando `git status`:

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:      index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

Todo aquello que sea conflictivo y no se haya podido resolver, se marca como "sin fusionar" (unmerged). Git añade a los archivos conflictivos unos marcadores especiales de resolución de conflictos que te guiarán cuando abras manualmente los archivos implicados y los edites para corregirlos. El archivo conflictivo contendrá algo como:

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
 please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

Donde nos dice que la versión en HEAD (la rama `master`, la que habías activado antes de lanzar el comando de fusión) contiene lo indicado en la parte superior del bloque (todo lo que está encima de `=======`) y que la versión en `iss53` contiene el resto, lo indicado en la parte inferior del bloque. Para resolver el conflicto, has de elegir manualmente el contenido de uno o de otro lado. Por ejemplo, puedes optar por cambiar el bloque, dejándolo así:

```
<div id="footer">
please contact us at email.support@github.com
</div>
```

Esta corrección contiene un poco de ambas partes y se han eliminado completamente las líneas `<<<<<<<` , `=======` y `>>>>>>>`. Tras resolver todos los bloques conflictivos, has de lanzar comandos `git add` para marcar cada archivo modificado. Marcar archivos como preparados (staged) indica a Git que sus conflictos han sido resueltos.

Si en lugar de resolver directamente prefieres utilizar una herramienta gráfica, puedes usar el comando `git mergetool`, el cual arrancará la correspondiente herramienta de visualización y te permitirá ir resolviendo conflictos con ella:

```
$ git mergetool

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge
p4merge araxis bc3 codecompare vimdiff emerge
Merging:
index.html

Normal merge conflict for 'index.html':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (opendiff):
```

Si deseas usar una herramienta distinta de la escogida por defecto (en mi caso `opendiff`, porque estoy lanzando el comando en Mac), puedes escogerla entre la lista de herramientas soportadas mostradas al principio ("merge tool candidates") tecleando el nombre de dicha herramienta.

NOTE Si necesitas herramientas más avanzadas para resolver conflictos de fusión más complicados, revisa la sección de fusionado en Fusión Avanzada.

Tras salir de la herramienta de fusionado, Git preguntará si hemos resuelto todos los conflictos y la fusión ha sido satisfactoria. Si le indicas que así ha sido, Git marca como preparado (staged) el archivo que acabamos de modificar. En cualquier momento, puedes lanzar el comando `git status` para ver si ya has resuelto todos los conflictos:

```
$ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

    modified:   index.html
```

Si todo ha ido correctamente, y ves que todos los archivos conflictivos están marcados como preparados, puedes lanzar el comando `git commit` para terminar de confirmar la fusión. El mensaje de confirmación por defecto será algo parecido a:

```
Merge branch 'iss53'

Conflicts:
    index.html
#
# It looks like you may be committing a merge.
# If this is not correct, please remove the file
#   .git/MERGE_HEAD
# and try again.


# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# All conflicts fixed but you are still merging.
#
# Changes to be committed:
#   modified:   index.html
#
```

Puedes modificar este mensaje añadiendo detalles sobre cómo has resuelto la fusión, si lo consideras útil para que otros entiendan esta fusión en un futuro. Se trata de indicar por qué has hecho lo que has hecho; a no ser que resulte obvio, claro está.

Gestión de Ramas

Ahora que ya has creado, fusionado y borrado algunas ramas, vamos a dar un vistazo a algunas herramientas de gestión muy útiles cuando comienzas a utilizar ramas de manera avanzada.

El comando `git branch` tiene más funciones que las de crear y borrar ramas. Si lo lanzas sin parámetros, obtienes una lista de las ramas presentes en tu proyecto:

```
$ git branch
  iss53
* master
  testing
```

Fijate en el carácter `*` delante de la rama `master`: nos indica la rama activa en este momento (la rama a la que apunta `HEAD`). Si hacemos una confirmación de cambios (commit), esa será la rama que avance. Para ver la última confirmación de cambios en cada rama, puedes usar el comando `git branch -v`:

```
$ git branch -v
  iss53   93b412c fix javascript issue
* master  7a98805 Merge branch 'iss53'
  testing 782fd34 add scott to the author list in the readmes
```

Otra opción útil para averiguar el estado de las ramas, es filtrarlas y mostrar solo aquellas que han sido fusionadas (o que no lo han sido) con la rama actualmente activa. Para ello, Git dispone de las opciones `--merged` y `--no-merged`. Si deseas ver las ramas que han sido fusionadas en la rama activa, puedes lanzar el comando `git branch --merged`:

```
$ git branch --merged
  iss53
* master
```

Aparece la rama `iss53` porque ya ha sido fusionada. Las ramas que no llevan por delante el carácter `*` pueden ser eliminadas sin problemas, porque todo su contenido ya ha sido incorporado a otras ramas.

Para mostrar todas las ramas que contienen trabajos sin fusionar, puedes utilizar el comando `git branch --no-merged`:

```
$ git branch --no-merged
  testing
```

Esto nos muestra la otra rama del proyecto. Debido a que contiene trabajos sin fusionar, al intentarla borrarla con `git branch -d`, el comando nos dará un error:

```
$ git branch -d testing
error: The branch 'testing' is not fully merged.
If you are sure you want to delete it, run 'git branch -D testing'.
```

Si realmente deseas borrar la rama, y perder el trabajo contenido en ella, puedes forzar el borrado con la opción `-D`; tal y como indica el mensaje de ayuda.

Flujos de Trabajo Ramificados

Ahora que ya has visto los procedimientos básicos de ramificación y fusión, ¿qué puedes o qué debes hacer con ellos? En este apartado vamos a ver algunos de los flujos de trabajo más comunes, de tal forma que puedas decidir si te gustaría incorporar alguno de ellos a tu ciclo de desarrollo.

Ramas de Largo Recorrido

Por la sencillez de la fusión a tres bandas de Git, el fusionar una rama a otra varias veces a lo largo del tiempo es fácil de hacer. Esto te posibilita tener varias ramas siempre abiertas, e irlas usando en diferentes etapas del ciclo de desarrollo; realizando fusiones frecuentes entre ellas.

Muchos desarrolladores que usan Git llevan un flujo de trabajo de esta naturaleza, manteniendo en la rama `master` únicamente el código totalmente estable (el código que ha sido o que va a ser liberado) y teniendo otras ramas paralelas denominadas `desarrollo` o `siguiente`, en las que trabajan y realizan pruebas. Estas ramas paralelas no suele estar siempre en un estado estable; pero cada vez que sí lo están, pueden ser fusionadas con la rama `master`. También es habitual el incorporarle

(pull) ramas puntuales (ramas temporales, como la rama `iss53` del ejemplo anterior) cuando las completamos y estamos seguros de que no van a introducir errores.

En realidad, en todo momento estamos hablando simplemente de apuntadores moviéndose por la línea temporal de confirmaciones de cambio (commit history). Las ramas estables apuntan hacia posiciones más antiguas en el historial de confirmaciones, mientras que las ramas avanzadas, las que van abriendo camino, apuntan hacia posiciones más recientes.

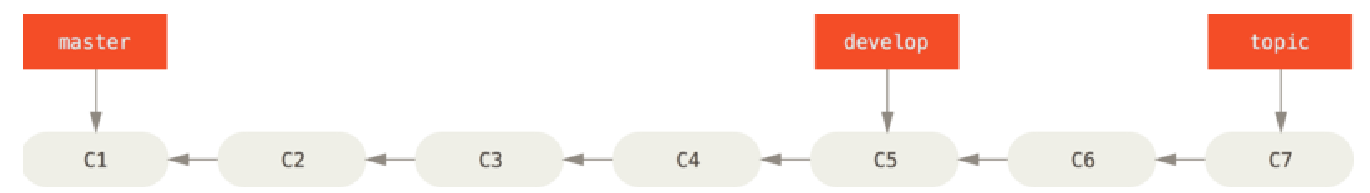


Figure 26. Una vista lineal del ramificado progresivo estable

Podría ser más sencillo pensar en las ramas como si fueran silos de almacenamiento, donde grupos de confirmaciones de cambio (commits) van siendo promocionados hacia silos más estables a medida que son probados y depurados.

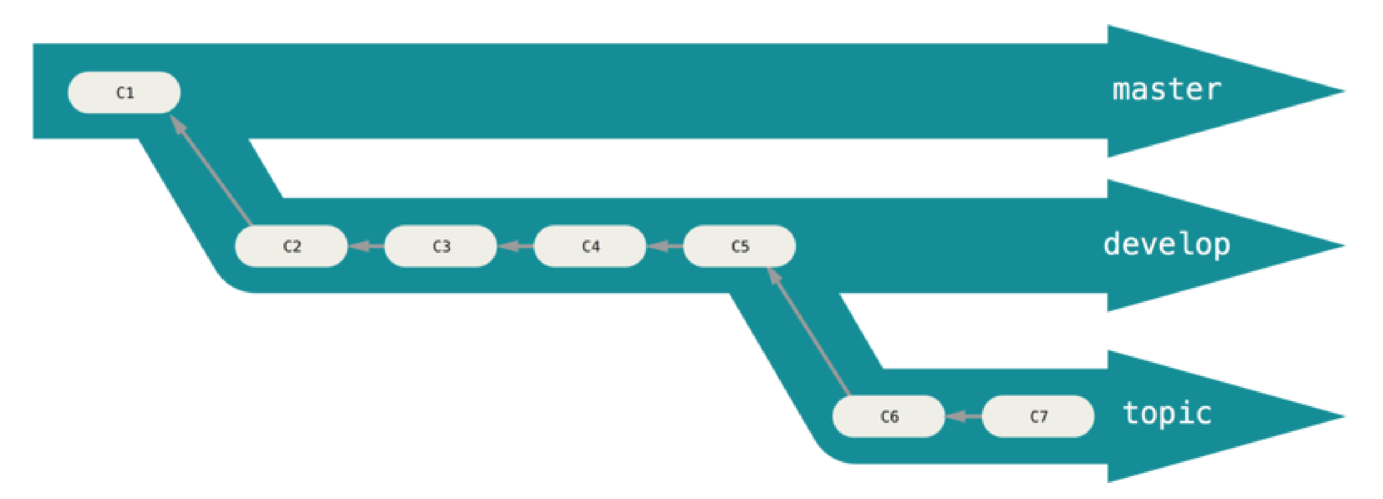


Figure 27. Una vista tipo “silo” del ramificado progresivo estable

Este sistema de trabajo se puede ampliar para diversos grados de estabilidad. Algunos proyectos muy grandes suelen tener una rama denominada `propuestas` o `pu` (del inglés “proposed updates”, propuesta de actualización), donde suele estar todo aquello integrado desde otras ramas, pero que aún no está listo para ser incorporado a las ramas `siguiente` o `master`. La idea es mantener siempre diversas ramas en diversos grados de estabilidad; pero cuando alguna alcanza un estado más estable, la fusionamos con la rama inmediatamente superior a ella. Aunque no es obligatorio el trabajar con ramas de larga duración, realmente es práctico y útil, sobre todo en proyectos largos o complejos.

Ramas Puntuales

Las ramas puntuales, en cambio, son útiles en proyectos de cualquier tamaño. Una rama puntual es aquella rama de corta duración que abres para un tema o para una funcionalidad determinada. Es algo que nunca habrías hecho en otro sistema VCS, debido a los altos costos de crear y fusionar ramas en esos sistemas. Pero en Git, por el contrario, es muy habitual el crear, trabajar con, fusionar y eliminar ramas varias veces al día.

Tal y como has visto con las ramas `iss53` y `hotfix` que has creado en la sección anterior. Has hecho algunas confirmaciones de cambio en ellas, y luego las has borrado tras fusionarlas con la rama principal. Esta técnica te posibilita realizar cambios de contexto rápidos y completos y, debido a que el trabajo está claramente separado en silos, con todos los cambios de cada tema en su propia rama, te será mucho más sencillo revisar el código y seguir su evolución. Puedes mantener los cambios ahí durante minutos, días o meses; y fusionarlos cuando realmente estén listos, sin importar el orden en el que fueron creados o en el que comenzaste a trabajar en ellos.

Por ejemplo, puedes realizar cierto trabajo en la rama `master`, ramificar para un problema concreto (rama `iss91`), trabajar en él un rato, ramificar una segunda vez para probar otra manera de resolverlo (rama `iss92v2`), volver a la rama `master` y trabajar un poco más, y, por último, ramificar temporalmente para probar algo de lo que no estás seguro (rama `dumbidea`). El historial de confirmaciones (commit history) será algo parecido esto:

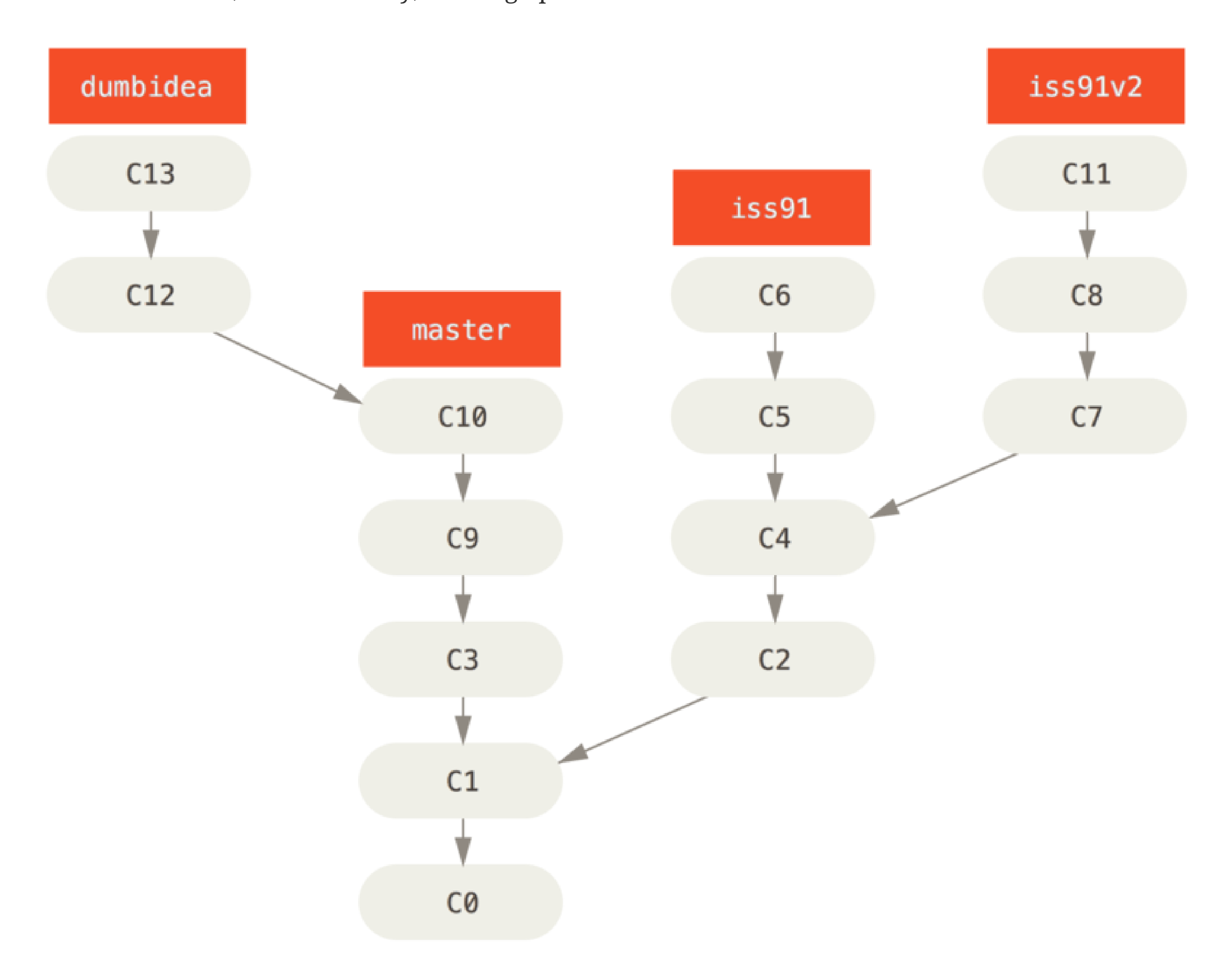


Figure 28. Múltiples ramas puntuales

En este momento, supongamos que te decides por la segunda solución al problema (rama `iss92v2`); y que, tras mostrar la rama `dumbidea` a tus compañeros, resulta que les parece una idea genial. Puedes descartar la rama `iss91` (perdiendo las confirmaciones C5 y C6), y fusionar las otras dos. El historial será algo parecido a esto:

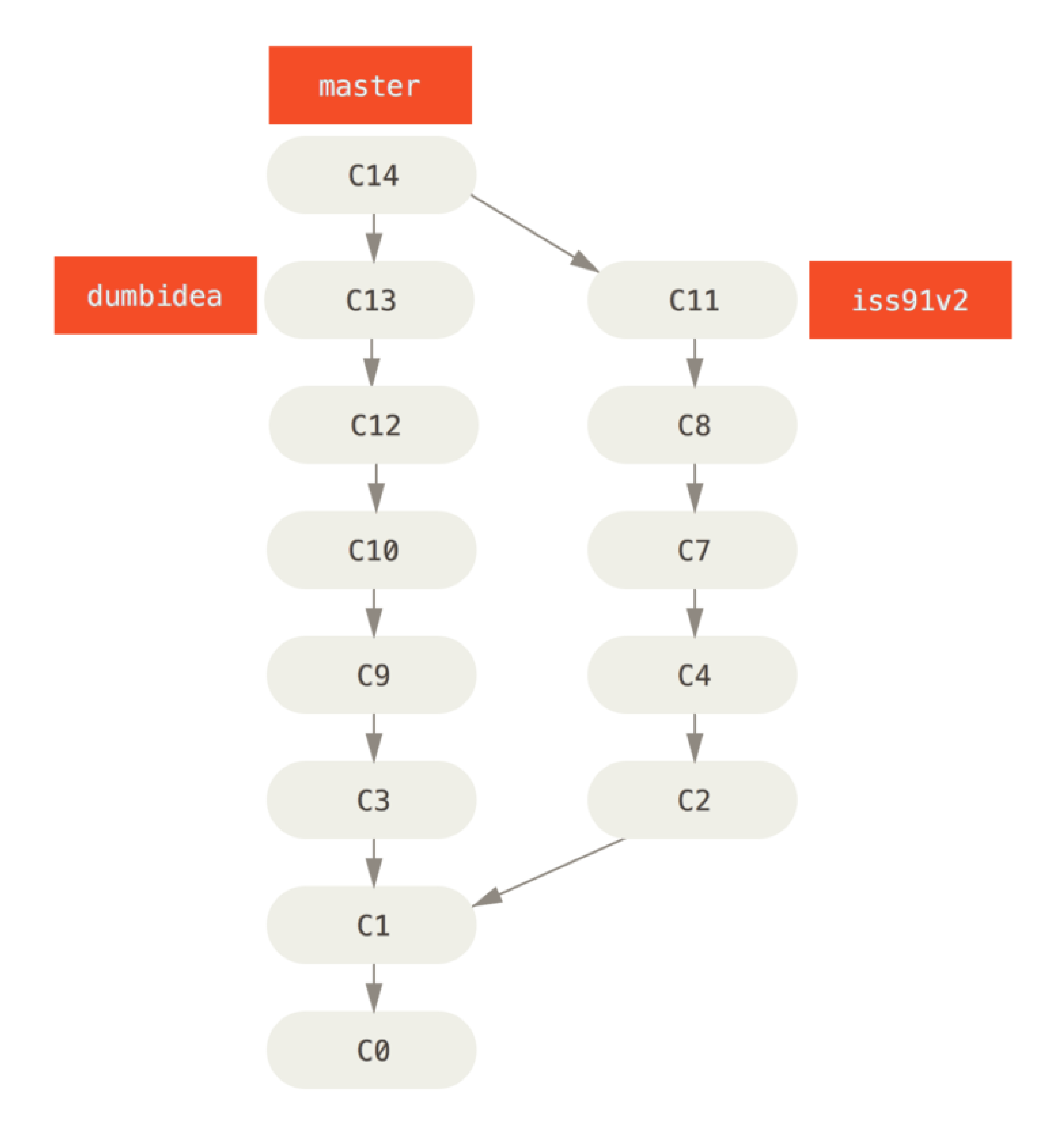


Figure 29. El historial tras fusionar `dumbidea` *e* `iss91v2`

Hablaremos un poco más sobre los distintos flujos de trabajo de tu proyecto Git en Git en entornos distribuidos, así que antes de decidir qué estilo de ramificación usará tu próximo proyecto, asegúrate de haber leído ese capítulo.

Es importante recordar que, mientras estás haciendo todo esto, todas las ramas son completamente locales. Cuando ramificas y fusionas, todo se realiza en tu propio repositorio Git. No hay ningún tipo de comunicación con ningún servidor.

Ramas Remotas

Las ramas remotas son referencias al estado de las ramas en tus repositorios remotos. Son ramas

locales que no puedes mover; se mueven automáticamente cuando estableces comunicaciones en la red. Las ramas remotas funcionan como marcadores, para recordarte en qué estado se encontraban tus repositorios remotos la última vez que conectaste con ellos.

Suelen referenciarse como `(remoto)/(rama)`. Por ejemplo, si quieres saber cómo estaba la rama `master` en el remoto `origin`, puedes revisar la rama `origin/master`. O si estás trabajando en un problema con un compañero y este envía (push) una rama `iss53`, tú tendrás tu propia rama de trabajo local `iss53`; pero la rama en el servidor apuntará a la última confirmación (commit) en la rama `origin/iss53`.

Esto puede ser un tanto confuso, pero intentemos aclararlo con un ejemplo. Supongamos que tienes un servidor Git en tu red, en `git.ourcompany.com`. Si haces un clon desde ahí, Git automáticamente lo denominará `origin`, traerá (pull) sus datos, creará un apuntador hacia donde esté en ese momento su rama `master` y denominará la copia local `origin/master`. Git te proporcionará también tu propia rama `master`, apuntando al mismo lugar que la rama `master` de `origin`; de manera que tengas donde trabajar.

NOTE

“origin” no es especial

Así como la rama “master” no tiene ningún significado especial en Git, tampoco lo tiene “origin”. “master” es un nombre muy usado solo porque es el nombre por defecto que Git le da a la rama inicial cuando ejecutas `git init`. De la misma manera, “origin” es el nombre por defecto que Git le da a un remoto cuando ejecutas `git clone`. Si en cambio ejecutases `git clone -o booyah`, tendrías una rama `booyah/master` como rama remota por defecto.

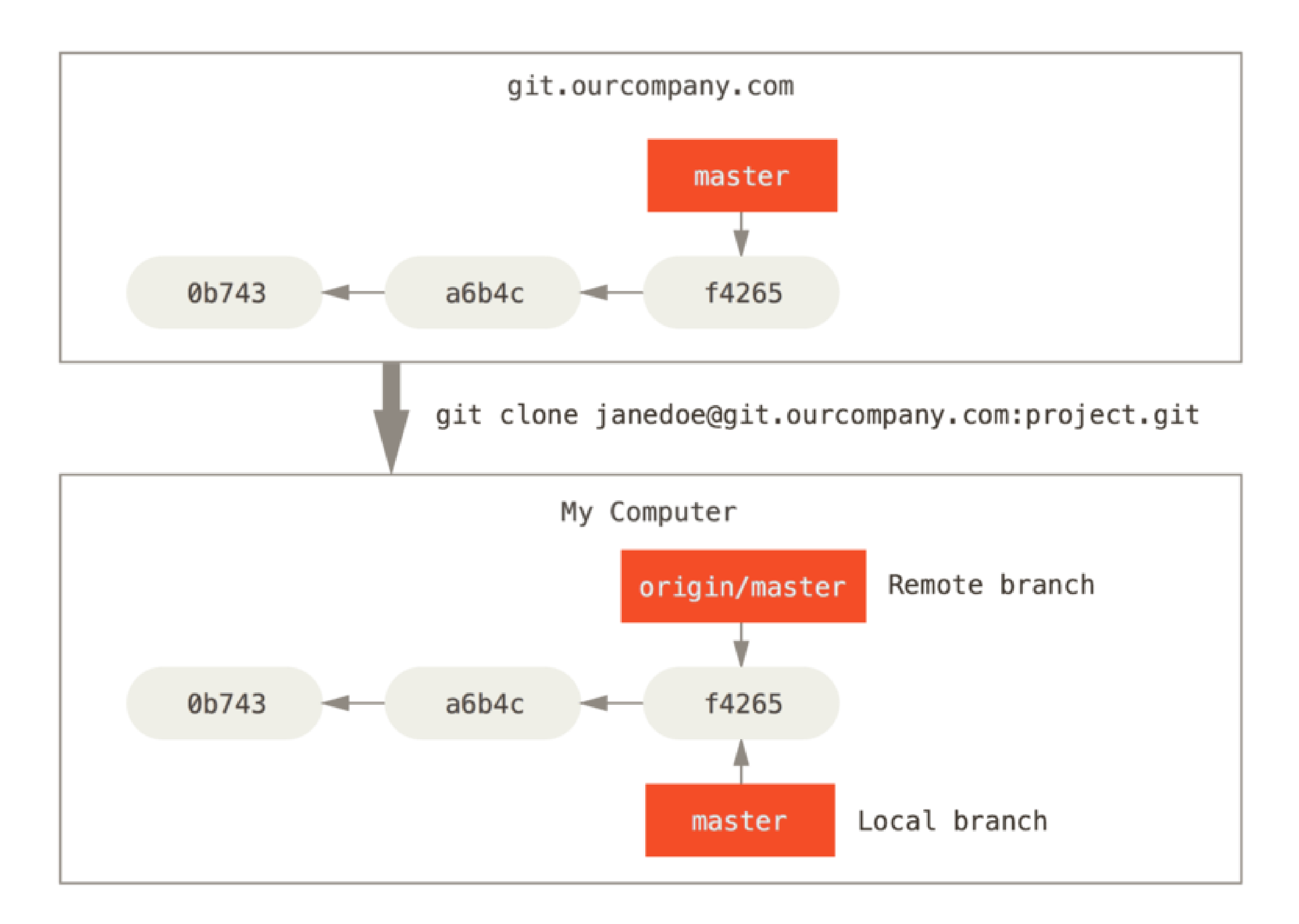
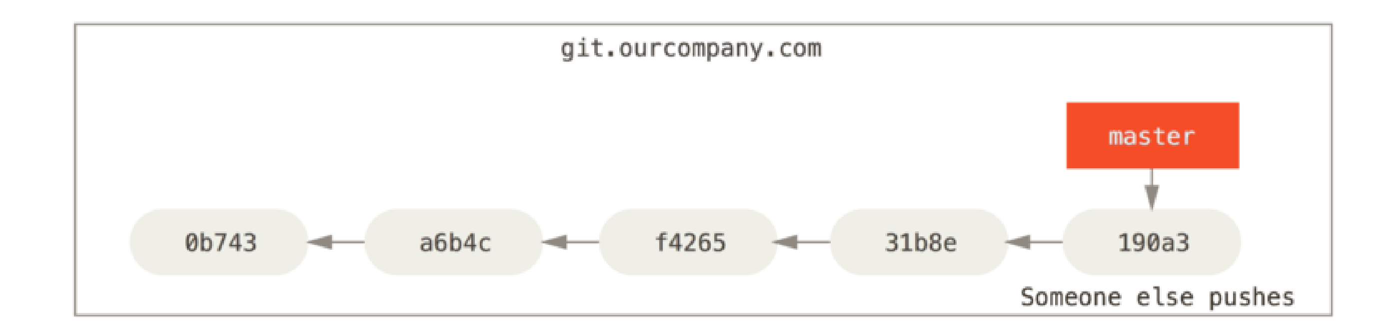


Figure 30. Servidor y repositorio local luego de ser clonado

Si haces algún trabajo en tu rama `master` local, y al mismo tiempo, alguien más lleva (push) su trabajo al servidor `git.ourcompany.com`, actualizando la rama `master` de allí, te encontrarás con que ambos registros avanzan de forma diferente. Además, mientras no tengas contacto con el servidor, tu apuntador a tu rama `origin/master` no se moverá.



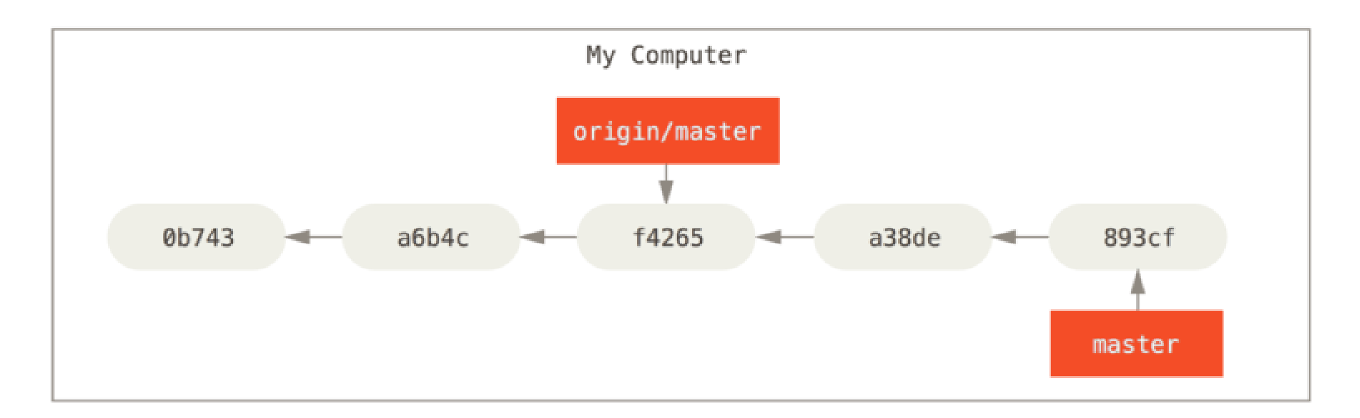


Figure 31. El trabajo remoto y el local pueden diverger

Para sincronizarte, puedes utilizar el comando `git fetch origin`. Este comando localiza en qué servidor está el origen (en este caso `git.ourcompany.com`), recupera cualquier dato presente allí que tú no tengas, y actualiza tu base de datos local, moviendo tu rama `origin/master` para que apunte a la posición más reciente.

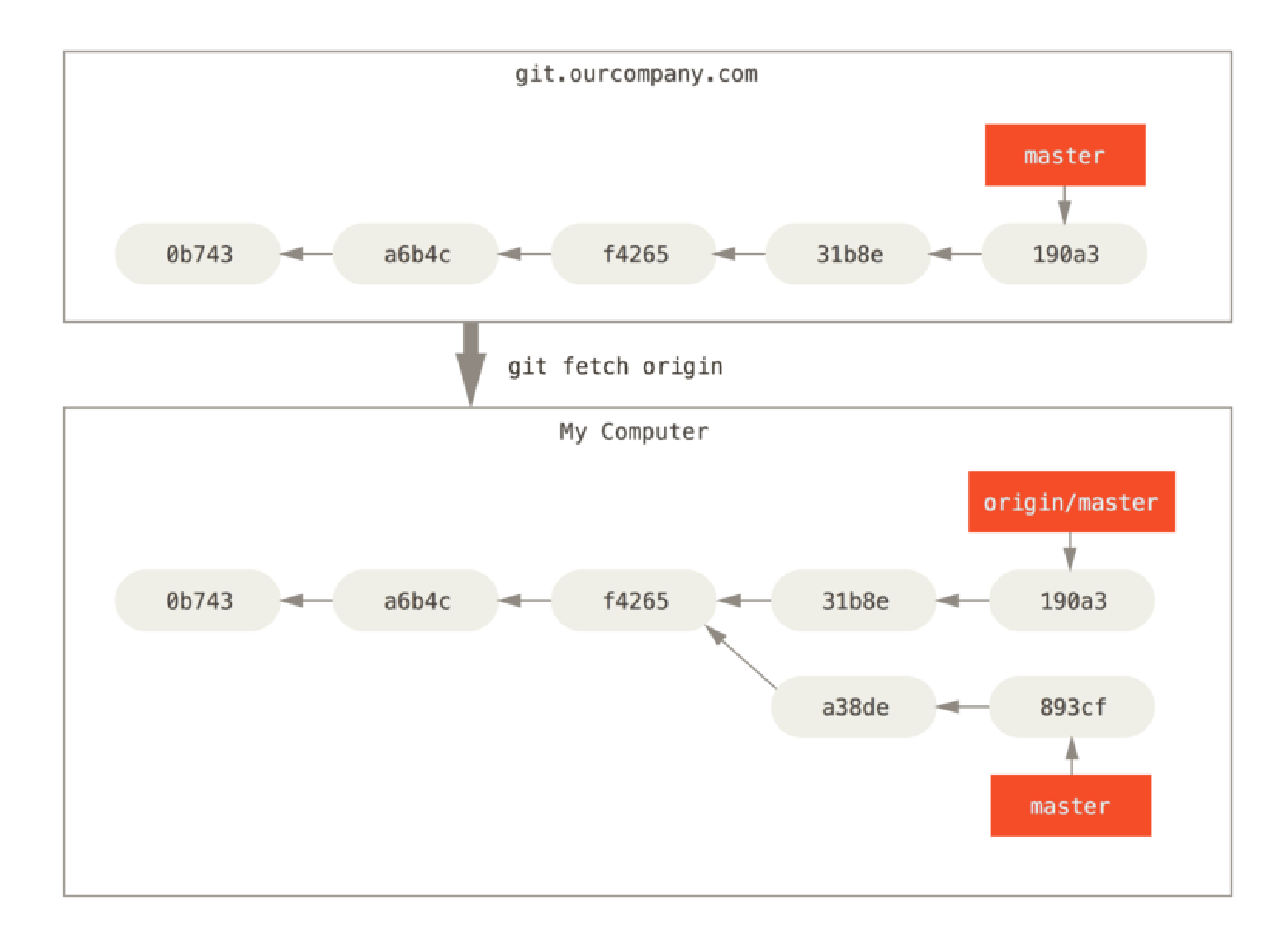
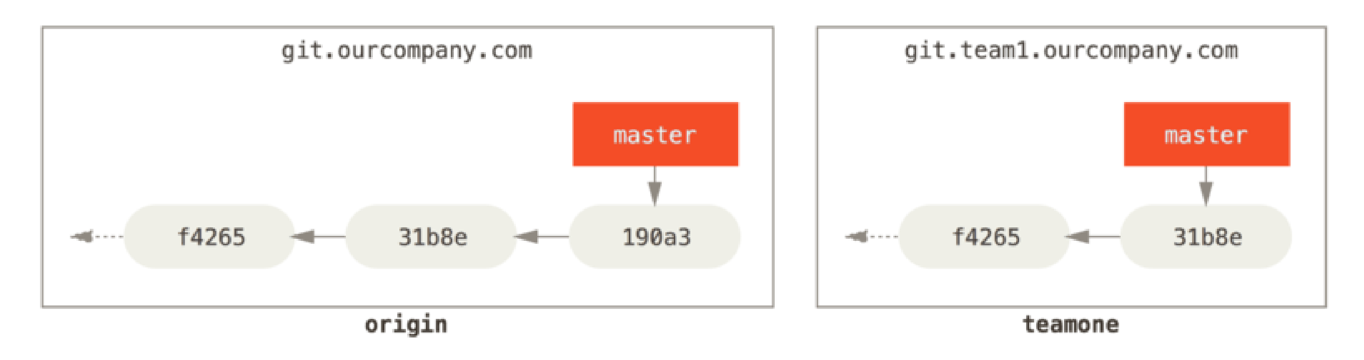


Figure 32. `git fetch` actualiza las referencias de tu remoto

Para ilustrar mejor el caso de tener múltiples servidores y cómo van las ramas remotas para esos proyectos remotos, supongamos que tienes otro servidor Git; utilizado por uno de tus equipos sprint, solamente para desarrollo. Este servidor se encuentra en `git.team1.ourcompany.com`. Puedes incluirlo como una nueva referencia remota a tu proyecto actual, mediante el comando `git remote add`, tal y como vimos en Fundamentos de Git. Puedes denominar `teamone` a este remoto al asignarle este nombre a la URL.



git remote add teamone git://git.team1.ourcompany.com

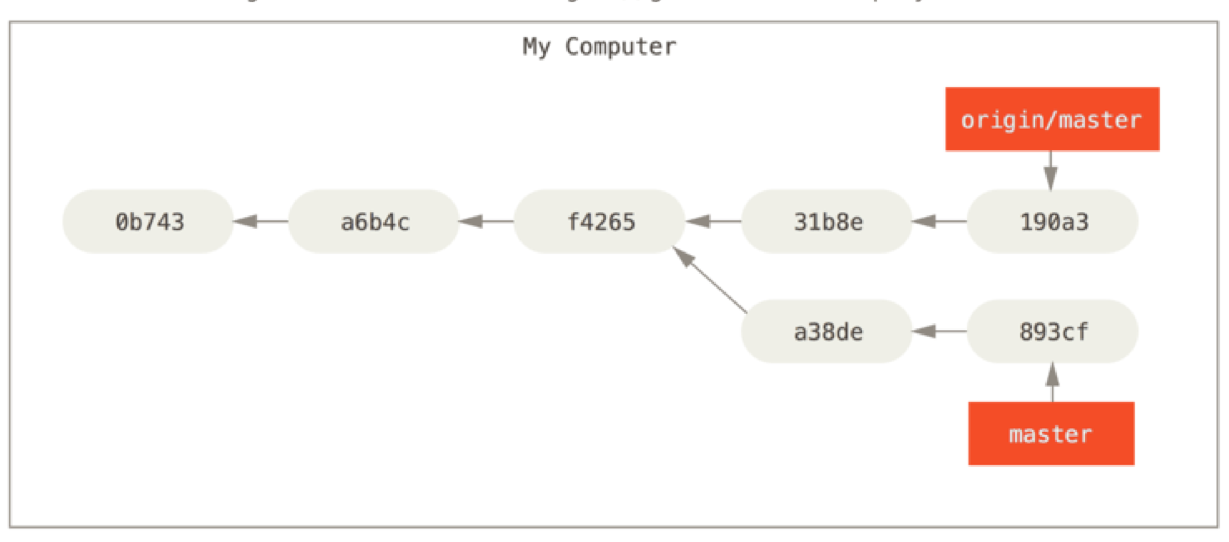


Figure 33. Añadiendo otro servidor como remoto

Ahora, puedes usar el comando `git fetch teamone` para recuperar todo el contenido del remoto `teamone` que tú no tenias. Debido a que dicho servidor es un subconjunto de los datos del servidor `origin` que tienes actualmente, Git no recupera (fetch) ningún dato; simplemente prepara una rama remota llamada `teamone/master` para apuntar a la confirmación (commit) que `teamone` tiene en su rama `master`.

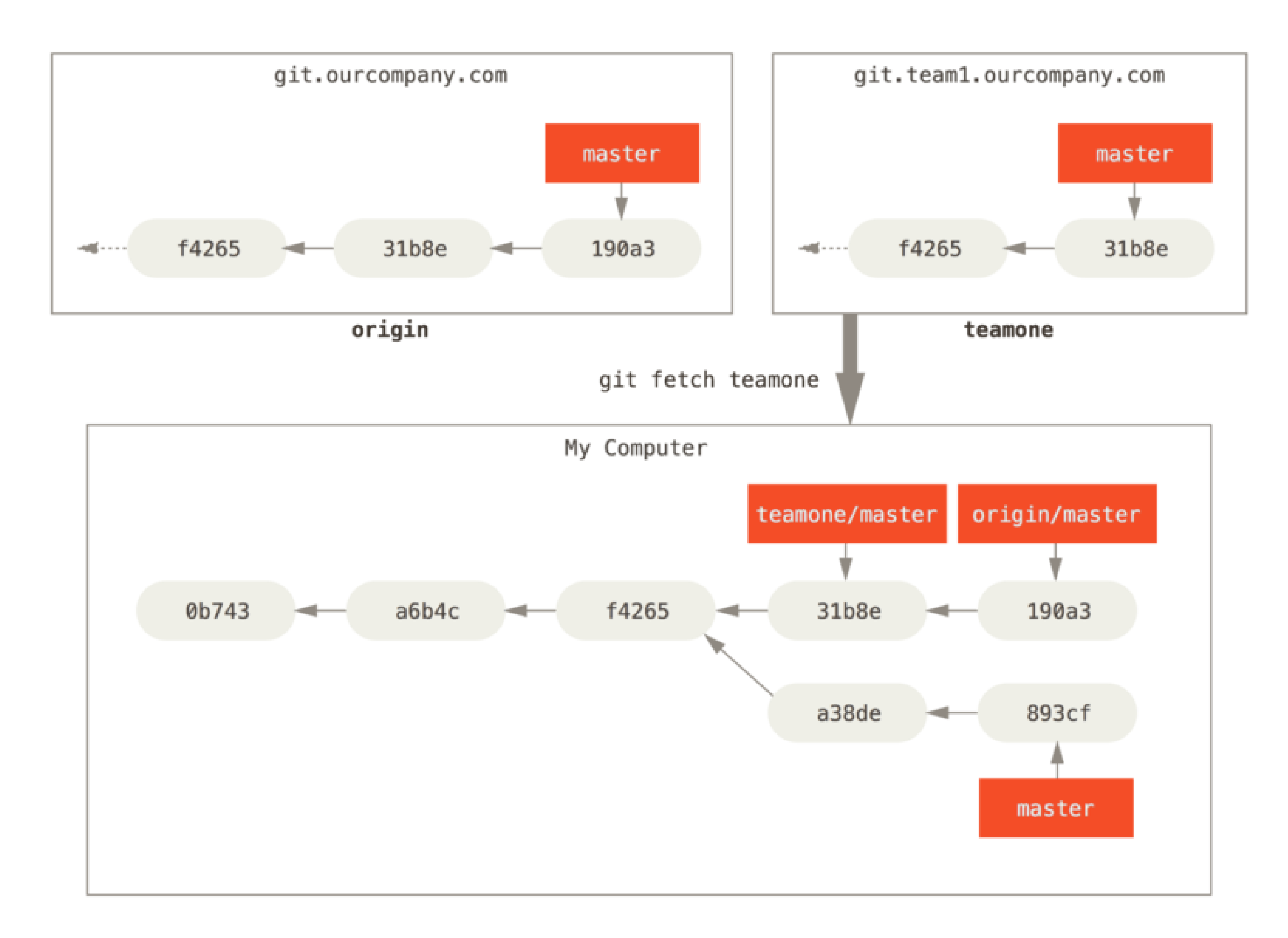


Figure 34. Seguimiento de la rama remota a través de `teamone/master`

Publicar

Cuando quieres compartir una rama con el resto del mundo, debes llevarla (push) a un remoto donde tengas permisos de escritura. Tus ramas locales no se sincronizan automáticamente con los remotos en los que escribes, sino que tienes que enviar (push) expresamente las ramas que desees compartir. De esta forma, puedes usar ramas privadas para el trabajo que no deseas compartir, llevando a un remoto tan solo aquellas partes que deseas aportar a los demás.

Si tienes una rama llamada `serverfix`, con la que vas a trabajar en colaboración; puedes llevarla al remoto de la misma forma que llevaste tu primera rama. Con el comando `git push (remoto) (rama)`:

```
$ git push origin serverfix
Counting objects: 24, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (24/24), 1.91 KiB | 0 bytes/s, done.
Total 24 (delta 2), reused 0 (delta 0)
To https://github.com/schacon/simplegit
 * [new branch]      serverfix -> serverfix
```

Esto es un atajo. Git expande automáticamente el nombre de rama `serverfix` a `refs/heads/serverfix:refs/heads/serverfix`, que significa: “coge mi rama local `serverfix` y actualiza con ella la rama `serverfix` del remoto”. Volveremos más tarde sobre el tema de `refs/heads/`,

viéndolo en detalle en Los entresijos internos de Git; por ahora, puedes ignorarlo. También puedes hacer `git push origin serverfix:serverfix`, que hace lo mismo; es decir: "coge mi `serverfix` y hazlo el `serverfix` remoto". Puedes utilizar este último formato para llevar una rama local a una rama remota con un nombre distinto. Si no quieres que se llame `serverfix` en el remoto, puedes lanzar, por ejemplo, `git push origin serverfix:awesomebranch`; para llevar tu rama `serverfix` local a la rama `awesomebranch` en el proyecto remoto.

> **NOTE**
>
> *No escribas tu contraseña todo el tiempo*
>
> Si utilizas una dirección URL con HTTPS para enviar datos, el servidor Git te preguntará tu usuario y contraseña para autenticarte. Por defecto, te pedirá esta información a través del terminal, para determinar si estás autorizado a enviar datos.
>
> Si no quieres escribir tu contraseña cada vez que haces un envío, puedes establecer un "cache de credenciales". La manera más sencilla de hacerlo es estableciéndolo en memoria por unos minutos, lo que puedes lograr fácilmente al ejecutar `git config --global credential.helper cache`
>
> Para más información sobre las distintas opciones de cache de credenciales, véase Almacenamiento de credenciales.

La próxima vez que tus colaboradores recuperen desde el servidor, obtendrán bajo la rama remota `origin/serverfix` una referencia a donde esté la versión de `serverfix` en el servidor:

```
$ git fetch origin
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/schacon/simplegit
 * [new branch]      serverfix    -> origin/serverfix
```

Es importante destacar que cuando recuperas (fetch) nuevas ramas remotas, no obtienes automáticamente una copia local editable de las mismas. En otras palabras, en este caso, no tienes una nueva rama `serverfix`. Sino que únicamente tienes un puntero no editable a `origin/serverfix`.

Para integrar (merge) esto en tu rama de trabajo actual, puedes usar el comando `git merge origin/serverfix`. Y si quieres tener tu propia rama `serverfix` para trabajar, puedes crearla directamente basandote en la rama remota:

```
$ git checkout -b serverfix origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

Esto sí te da una rama local donde puedes trabajar, que comienza donde `origin/serverfix` estaba en ese momento.

Hacer Seguimiento a las Ramas

Al activar (checkout) una rama local a partir de una rama remota, se crea automáticamente lo que podríamos denominar una “rama de seguimiento” (tracking branch). Las ramas de seguimiento son ramas locales que tienen una relación directa con alguna rama remota. Si estás en una rama de seguimiento y tecleas el comando `git pull`, Git sabe de cuál servidor recuperar (fetch) y fusionar (merge) datos.

Cuando clonas un repositorio, este suele crear automáticamente una rama `master` que hace seguimiento de `origin/master`. Sin embargo, puedes preparar otras ramas de seguimiento si deseas tener unas que sigan ramas de otros remotos o no seguir la rama `master`. El ejemplo más simple es el que acabas de ver al lanzar el comando `git checkout -b [rama] [nombreremoto]/[rama]`. Esta operación es tan común que git ofrece el parámetro `--track`:

```
$ git checkout --track origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

Para preparar una rama local con un nombre distinto a la del remoto, puedes utilizar la primera versión con un nombre de rama local diferente:

```
$ git checkout -b sf origin/serverfix
Branch sf set up to track remote branch serverfix from origin.
Switched to a new branch 'sf'
```

Así, tu rama local `sf` traerá (pull) información automáticamente desde `origin/serverfix`.

Si ya tienes una rama local y quieres asignarla a una rama remota que acabas de traerte, o quieres cambiar la rama a la que le haces seguimiento, puedes usar en cualquier momento las opciones `-u` o `--set-upstream-to` del comando `git branch`.

```
$ git branch -u origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
```

NOTE

Atajo al upstream

Cuando tienes asignada una rama de seguimiento, puedes hacer referencia a ella mediante `@{upstream}` o mediante el atajo `@{u}`. De esta manera, si estás en la rama `master` y esta sigue a la rama `origin/master`, puedes hacer algo como `git merge @{u}` en vez de `git merge origin/master`.

Si quieres ver las ramas de seguimiento que tienes asignado, puedes usar la opción `-vv` con `git branch`. Esto listará tus ramas locales con más información, incluyendo a qué sigue cada rama y si tu rama local está por delante, por detrás o ambas.

```
$ git branch -vv
  iss53     7e424c3 [origin/iss53: ahead 2] forgot the brackets
  master    1ae2a45 [origin/master] deploying index fix
* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] this should do it
  testing   5ea463a trying something new
```

Aquí podemos ver que nuestra rama `iss53` sigue `origin/iss53` y está "ahead" (delante) por dos, es decir, que tenemos dos confirmaciones locales que no han sido enviadas al servidor. También podemos ver que nuestra rama `master` sigue a `origin/master` y está actualizada. Luego podemos ver que nuestra rama `serverfix` sigue la rama `server-fix-good` de nuestro servidor `teamone` y que está tres cambios por delante (ahead) y uno por detrás (behind), lo que significa que existe una confirmación en el servidor que no hemos fusionado y que tenemos tres confirmaciones locales que no hemos enviado. Por último, podemos ver que nuestra rama `testing` no sigue a ninguna rama remota.

Es importante destacar que estos números se refieren a la última vez que trajiste (fetch) datos de cada servidor. Este comando no se comunica con los servidores, solo te indica lo que sabe de ellos localmente. Si quieres tener los cambios por delante y por detrás actualizados, debes traértelos (fetch) de cada servidor antes de ejecutar el comando. Puedes hacerlo de esta manera: `$ git fetch --all; git branch -vv`

Traer y Fusionar

A pesar de que el comando `git fetch` trae todos los cambios del servidor que no tienes, este no modifica tu directorio de trabajo. Simplemente obtendrá los datos y dejará que tú mismo los fusiones. Sin embargo, existe un comando llamado `git pull`, el cuál básicamente hace `git fetch` seguido por `git merge` en la mayoría de los casos. Si tienes una rama de seguimiento configurada como vimos en la última sección, bien sea asignándola explícitamente o creándola mediante los comandos `clone` o `checkout`, `git pull` identificará a qué servidor y rama remota sigue tu rama actual, traerá los datos de dicho servidor e intentará fusionar dicha rama remota.

Normalmente es mejor usar los comandos `fetch` y `merge` de manera explícita pues la magia de `git pull` puede resultar confusa.

Eliminar Ramas Remotas

Imagina que ya has terminado con una rama remota, es decir, tanto tú como tus colaboradores habéis completado una determinada funcionalidad y la habéis incorporado (merge) a la rama `master` en el remoto (o donde quiera que tengáis la rama de código estable). Puedes borrar la rama remota utilizando la opción `--delete` de `git push`. Por ejemplo, si quieres borrar la rama `serverfix` del servidor, puedes utilizar:

```
$ git push origin --delete serverfix
To https://github.com/schacon/simplegit
 - [deleted]         serverfix
```

Básicamente lo que hace es eliminar el apuntador del servidor. El servidor Git suele mantener los

datos por un tiempo hasta que el recolector de basura se ejecute, de manera que si la has borrado accidentalmente, suele ser fácil recuperarla.

Reorganizar el Trabajo Realizado

En Git tenemos dos formas de integrar cambios de una rama en otra: la fusión (merge) y la reorganización (rebase). En esta sección vas a aprender en qué consiste la reorganización, cómo utilizarla, por qué es una herramienta sorprendente y en qué casos no es conveniente utilizarla.

Reorganización Básica

Volviendo al ejemplo anterior, en la sección sobre fusiones Procedimientos Básicos de Fusión puedes ver que has separado tu trabajo y realizado confirmaciones (commit) en dos ramas diferentes.

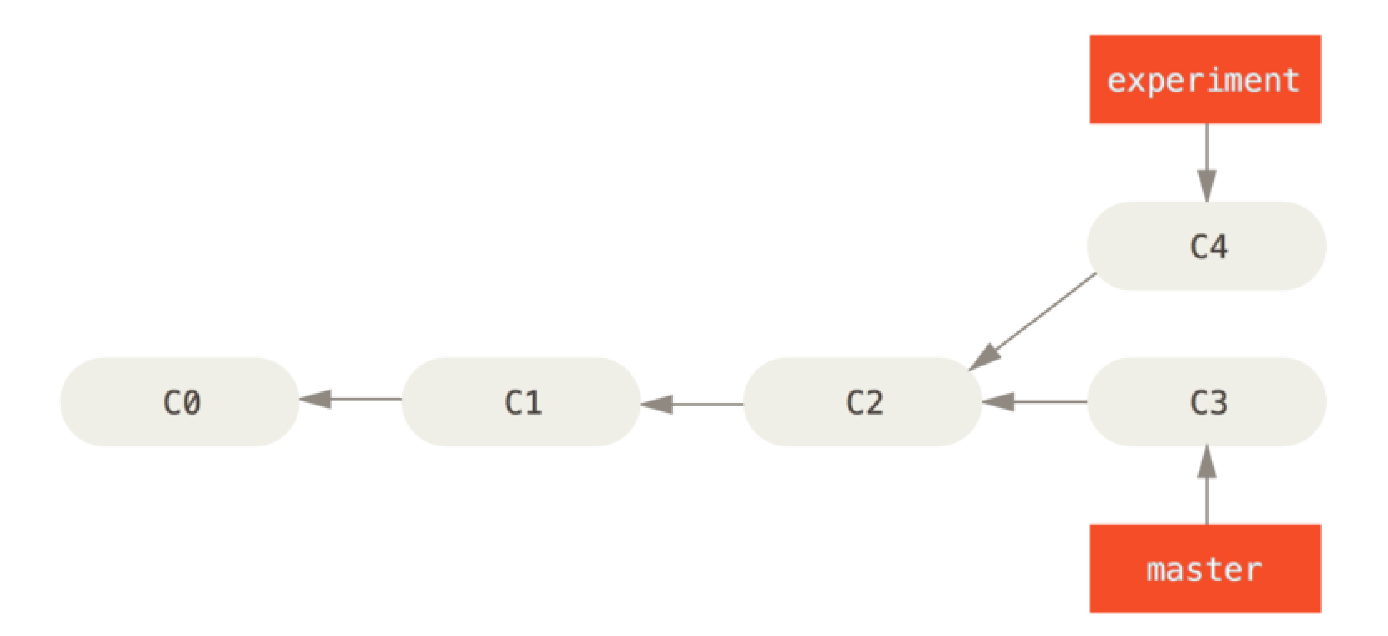


Figure 35. El registro de confirmaciones inicial

La manera más sencilla de integrar ramas, tal y como hemos visto, es el comando `git merge`. Realiza una fusión a tres bandas entre las dos últimas instantáneas de cada rama (C3 y C4) y el ancestro común a ambas (C2); creando una nueva instantánea (snapshot) y la correspondiente confirmación (commit).

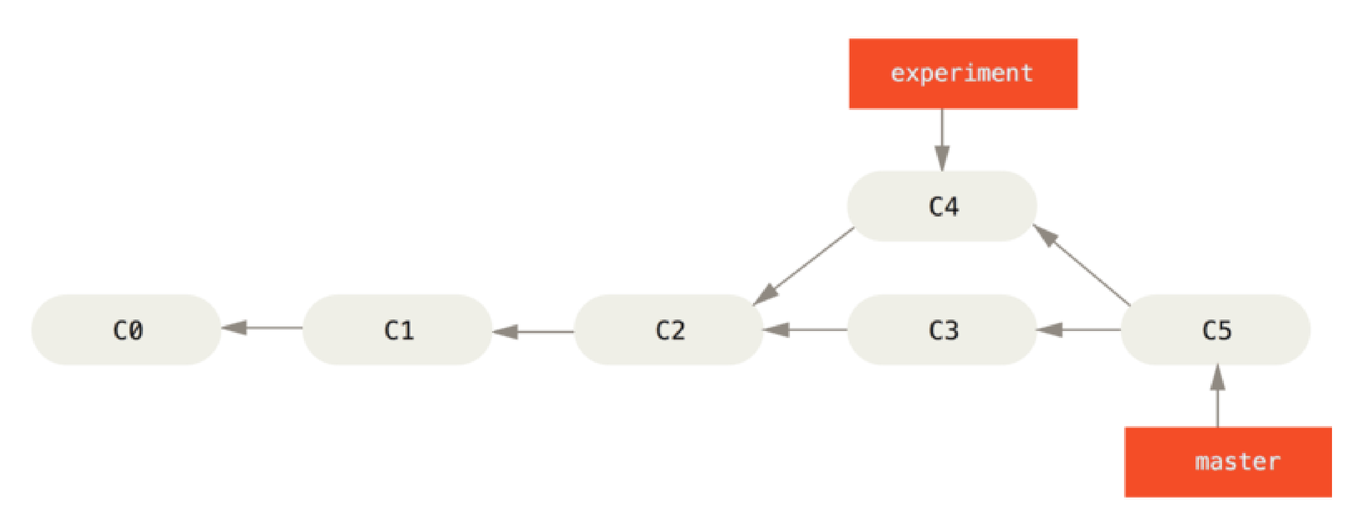


Figure 36. Fusionar una rama para integrar el registro de trabajos divergentes

Sin embargo, también hay otra forma de hacerlo: puedes coger los cambios introducidos en C3 y reaplicarlos encima de C4. Esto es lo que en Git llamamos *reorganizar* (*rebasing*, en inglés). Con el comando `git rebase`, puedes coger todos los cambios confirmados en una rama, y reaplicarlos sobre otra.

Por ejemplo, puedes lanzar los comandos:

```
$ git checkout experiment
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```

Haciendo que Git vaya al ancestro común de ambas ramas (donde estás actualmente y de donde quieres reorganizar), saque las diferencias introducidas por cada confirmación en la rama donde estás, guarde esas diferencias en archivos temporales, reinicie (reset) la rama actual hasta llevarla a la misma confirmación en la rama de donde quieres reorganizar, y, finalmente, vuelva a aplicar ordenadamente los cambios.

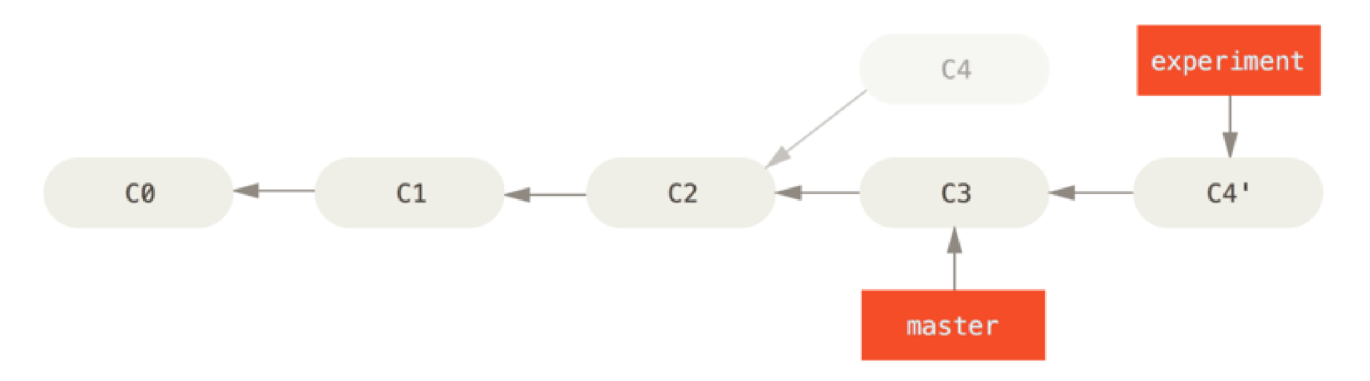


Figure 37. Reorganizando sobre C3 los cambios introducidos en C4

En este momento, puedes volver a la rama `master` y hacer una fusión con avance rápido (fast-forward merge).

```
$ git checkout master
$ git merge experiment
```

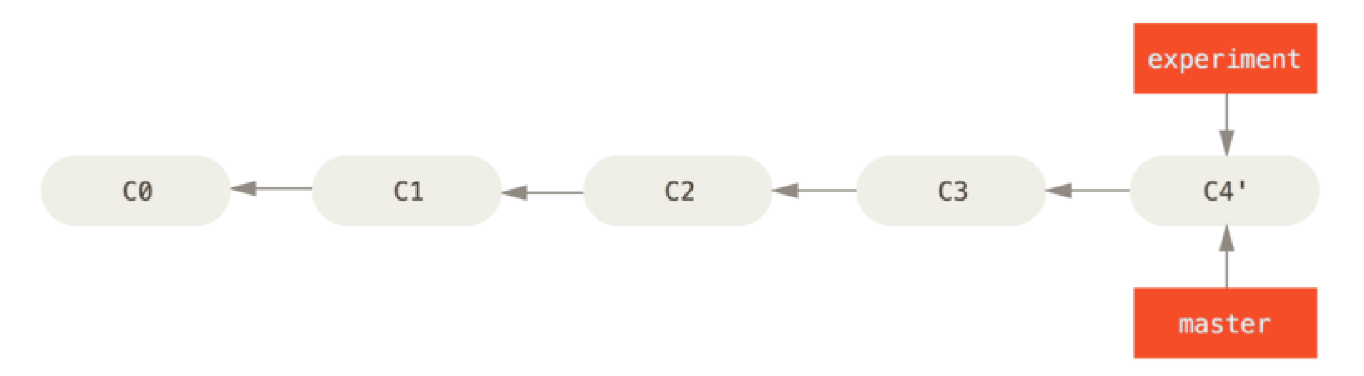


Figure 38. Avance rápido de la rama `master`

Así, la instantánea apuntada por `C4'` es exactamente la misma apuntada por `C5` en el ejemplo de la fusión. No hay ninguna diferencia en el resultado final de la integración, pero el haberla hecho reorganizando nos deja un historial más claro. Si examinas el historial de una rama reorganizada, este aparece siempre como un historial lineal: como si todo el trabajo se hubiera realizado en series, aunque realmente se haya hecho en paralelo.

Habitualmente, optarás por esta vía cuando quieras estar seguro de que tus confirmaciones de cambio (commits) se pueden aplicar limpiamente sobre una rama remota; posiblemente, en un proyecto donde estés intentando colaborar, pero lleves tú el mantenimiento. En casos como esos, puedes trabajar sobre una rama y luego reorganizar lo realizado en la rama `origin/master` cuando lo tengas todo listo para enviarlo al proyecto principal. De esta forma, la persona que mantiene el proyecto no necesitará hacer ninguna integración con tu trabajo; le bastará con un avance rápido o una incorporación limpia.

Cabe destacar que la instantánea (snapshot) apuntada por la confirmación (commit) final, tanto si es producto de una reorganización (rebase) como si lo es de una fusión (merge), es exactamente la misma instantánea; lo único diferente es el historial. La reorganización vuelve a aplicar cambios de una rama de trabajo sobre otra rama, en el mismo orden en que fueron introducidos en la primera, mientras que la fusión combina entre sí los dos puntos finales de ambas ramas.

Algunas Reorganizaciones Interesantes

También puedes aplicar una reorganización (rebase) sobre otra cosa además de sobre la rama de reorganización. Por ejemplo, considera un historial como el de Un historial con una rama puntual sobre otra rama puntual. Has ramificado a una rama puntual (`server`) para añadir algunas funcionalidades al proyecto, y luego has confirmado los cambios. Después, vuelves a la rama original para hacer algunos cambios en la parte cliente (rama `client`), y confirmas también esos cambios. Por último, vuelves sobre la rama `server` y haces algunos cambios más.

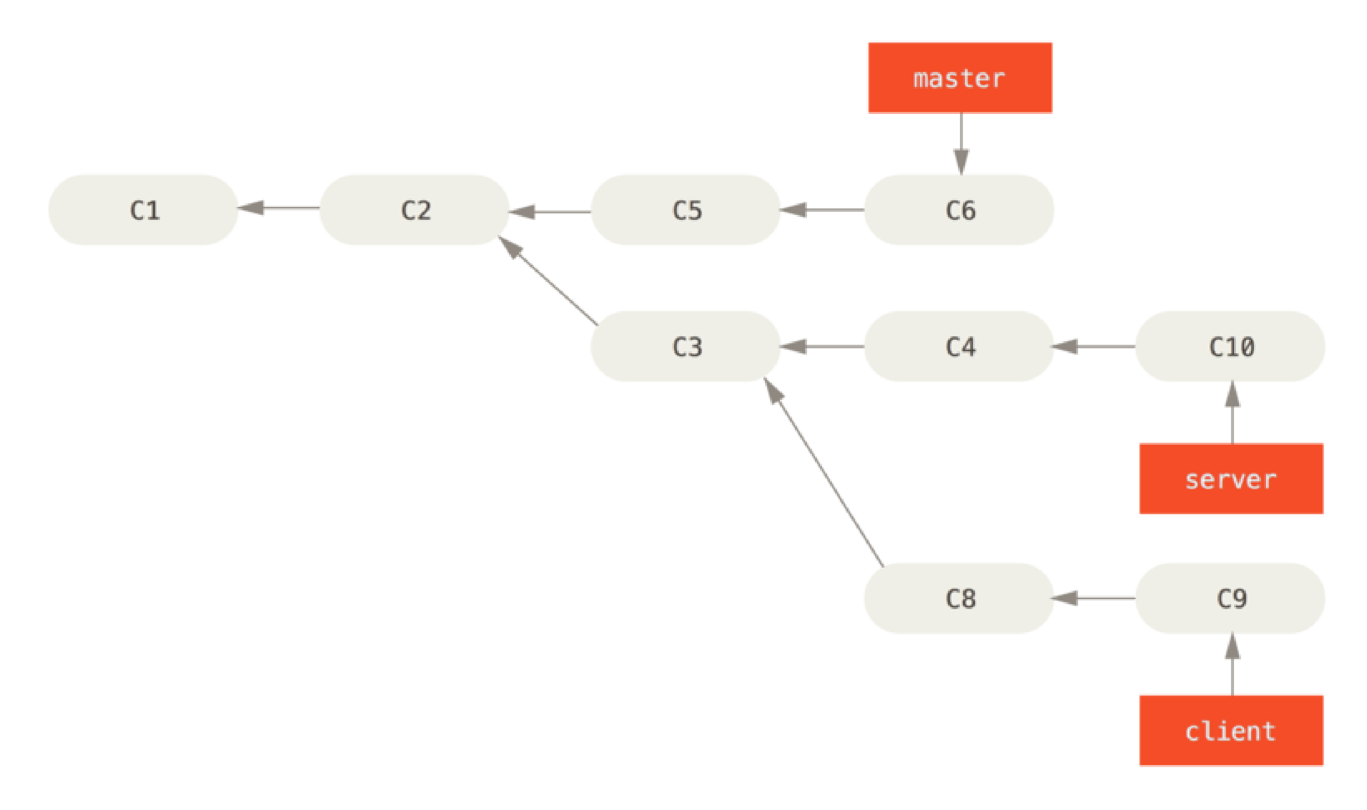


Figure 39. Un historial con una rama puntual sobre otra rama puntual

Imagina que decides incorporar tus cambios del lado cliente sobre el proyecto principal para hacer un lanzamiento de versión; pero no quieres lanzar aún los cambios del lado servidor porque no están aún suficientemente probados. Puedes coger los cambios del cliente que no están en server (`C8` y `C9`) y reaplicarlos sobre tu rama principal usando la opción `--onto` del comando `git rebase`:

```
$ git rebase --onto master server client
```

Esto viene a decir: "Activa la rama `client`, averigua los cambios desde el ancestro común entre las ramas `client` y `server`, y aplicalos en la rama `master`". Puede parecer un poco complicado, pero los resultados son realmente interesantes.

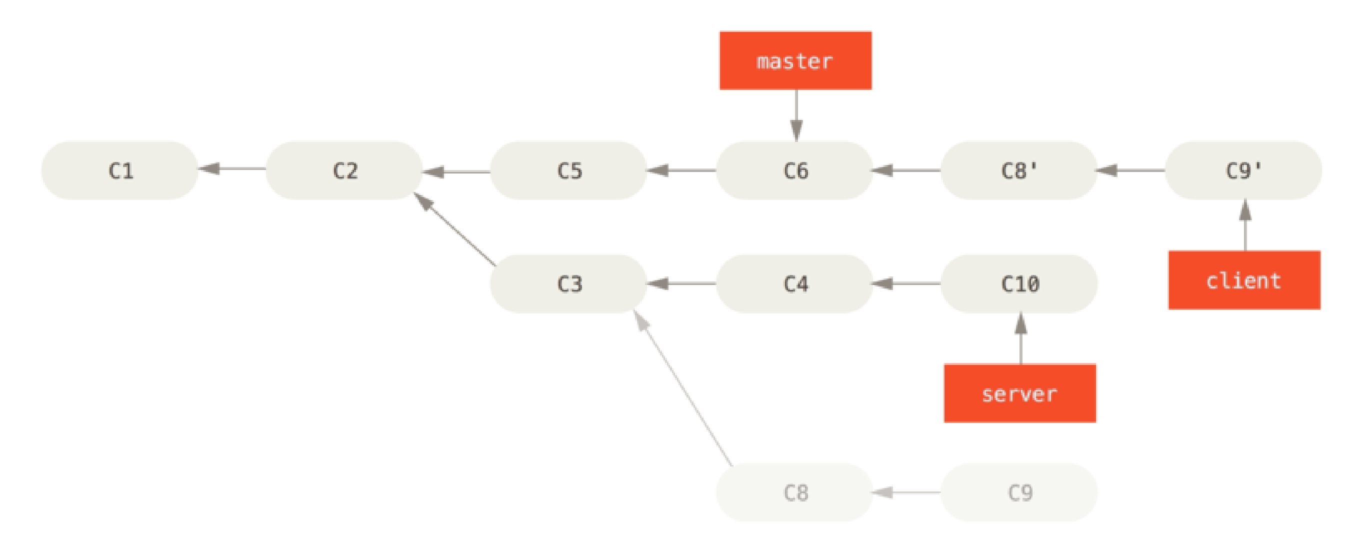


Figure 40. Reorganizando una rama puntual fuera de otra rama puntual

Y, tras esto, ya puedes avanzar la rama principal (ver Avance rápido de tu rama `master`, para incluir

los cambios de la rama `client`):

```
$ git checkout master
$ git merge client
```

Figure 41. Avance rápido de tu rama `master`, para incluir los cambios de la rama `client`

Ahora supongamos que decides traerlos (pull) también sobre tu rama `server`. Puedes reorganizar (rebase) la rama `server` sobre la rama `master` sin necesidad siquiera de comprobarlo previamente, usando el comando `git rebase [rama-base] [rama-puntual]`, el cual activa la rama puntual (`server` en este caso) y la aplica sobre la rama base (`master` en este caso):

```
$ git rebase master server
```

Esto vuelca el trabajo de `server` sobre el de `master`, tal y como se muestra en Reorganizando la rama `server` sobre la rama `master`.

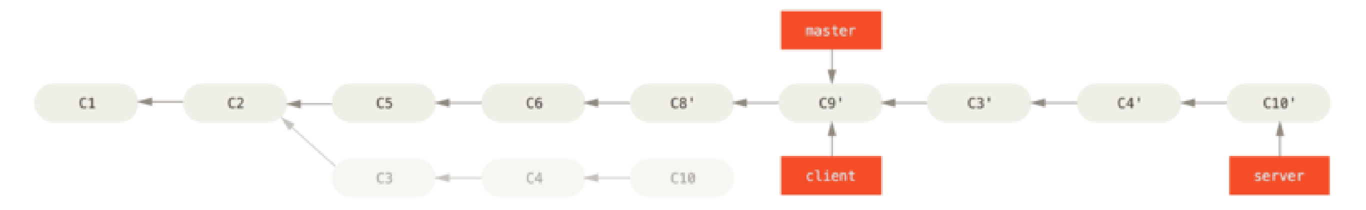


Figure 42. Reorganizando la rama `server` sobre la rama `master`

Después, puedes avanzar rápidamente la rama base (`master`):

```
$ git checkout master
$ git merge server
```

Y por último puedes eliminar las ramas `client` y `server` porque ya todo su contenido ha sido integrado y no las vas a necesitar más, dejando tu registro tras todo este proceso tal y como se muestra en Historial final de confirmaciones de cambio:

```
$ git branch -d client
$ git branch -d server
```

Figure 43. Historial final de confirmaciones de cambio

Los Peligros de Reorganizar

Ahh..., pero la dicha de la reorganización no la alcanzamos sin sus contrapartidas, las cuales

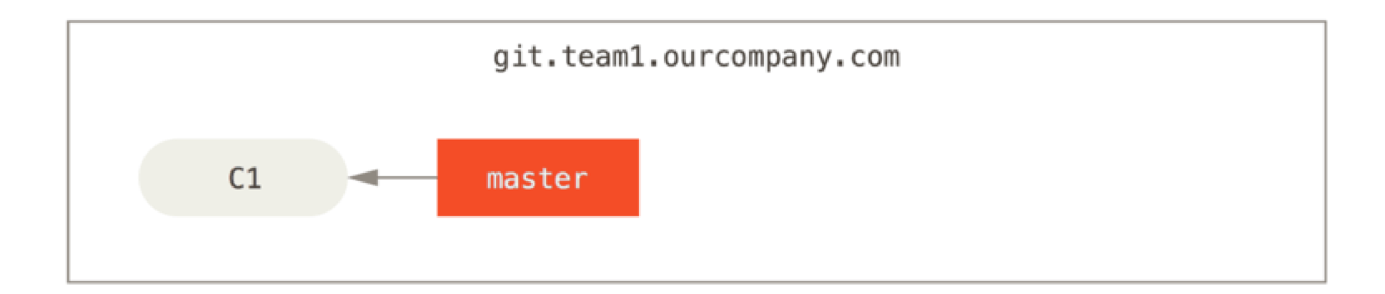
pueden resumirse en una línea:

Nunca reorganices confirmaciones de cambio (commits) que hayas enviado (push) a un repositorio público.

Si sigues esta recomendación, no tendrás problemas. Pero si no lo haces, la gente te odiará y serás despreciado por tus familiares y amigos.

Cuando reorganizas algo, estás abandonando las confirmaciones de cambio ya creadas y estás creando unas nuevas; que son similares, pero diferentes. Si envias (push) confirmaciones (commits) a alguna parte, y otros las recogen (pull) de allí; y después vas tú y las reescribes con `git rebase` y las vuelves a enviar (push); tus colaboradores tendrán que refusionar (re-merge) su trabajo y todo se volverá tremendamente complicado cuando intentes recoger (pull) su trabajo de vuelta sobre el tuyo.

Veamos con un ejemplo como reorganizar trabajo que has hecho público puede causar problemas. Imagínate que haces un clon desde un servidor central, y luego trabajas sobre él. Tu historial de cambios puede ser algo como esto:



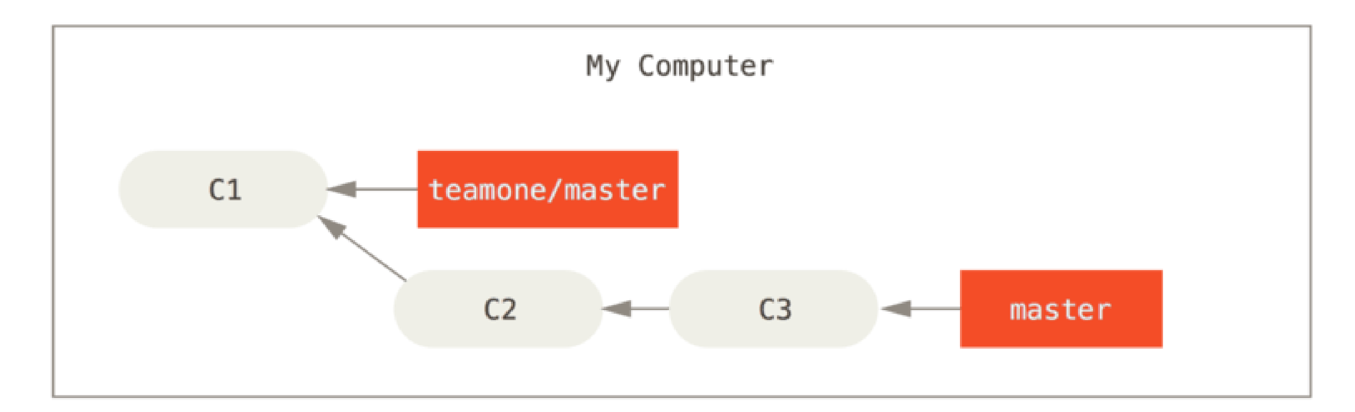
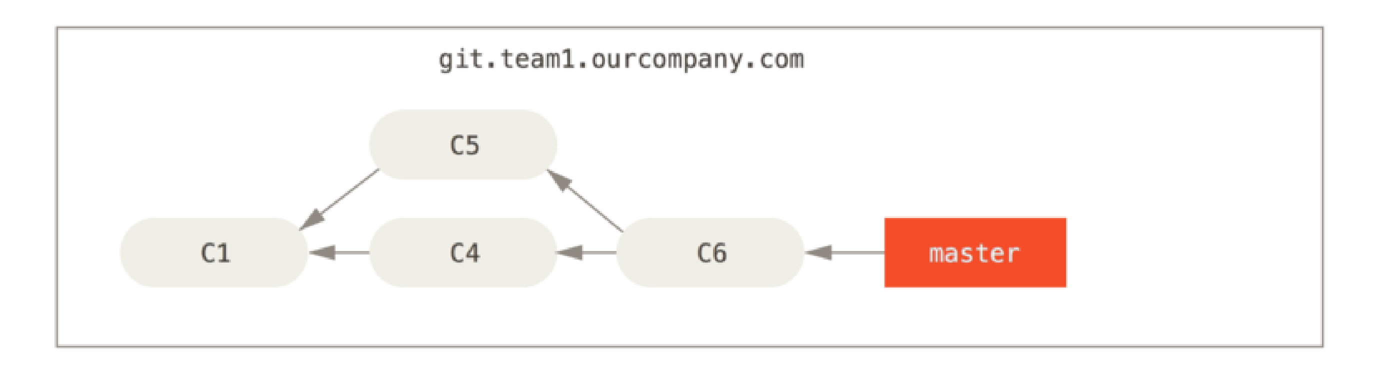


Figure 44. Clonar un repositorio y trabajar sobre él

Ahora, otra persona trabaja también sobre ello, realiza una fusión (merge) y lleva (push) su trabajo al servidor central. Tú te traes (fetch) sus trabajos y los fusionas (merge) sobre una nueva rama en tu trabajo, con lo que tu historial quedaría parecido a esto:



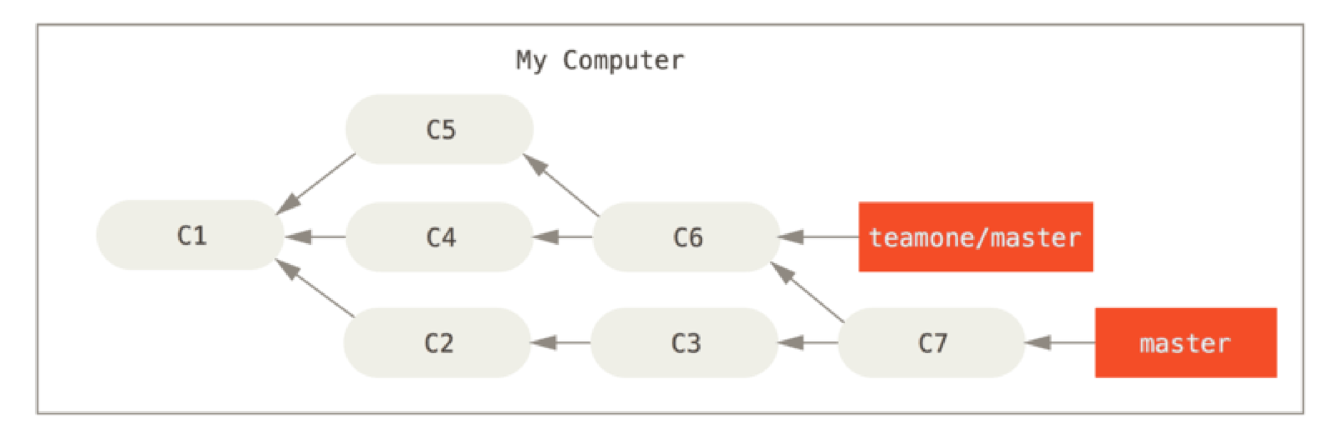
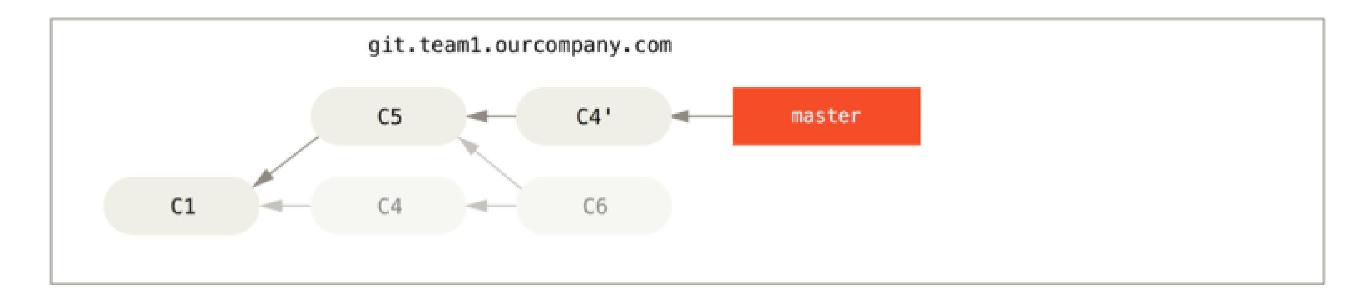


Figure 45. Traer (fetch) algunas confirmaciones de cambio (commits) y fusionarlas (merge) sobre tu trabajo

A continuación, la persona que había llevado cambios al servidor central decide retroceder y reorganizar su trabajo; haciendo un `git push --force` para sobrescribir el registro en el servidor. Tu te traes (fetch) esos nuevos cambios desde el servidor.

Figure 46. Alguien envio (push) confirmaciones (commits) reorganizadas, abandonando las confirmaciones en las que tu habías basado tu trabajo

Ahora los dos están en un aprieto. Si haces `git pull` crearás una fusión confirmada, la cual incluirá ambas líneas del historial, y tu repositorio lucirá así:



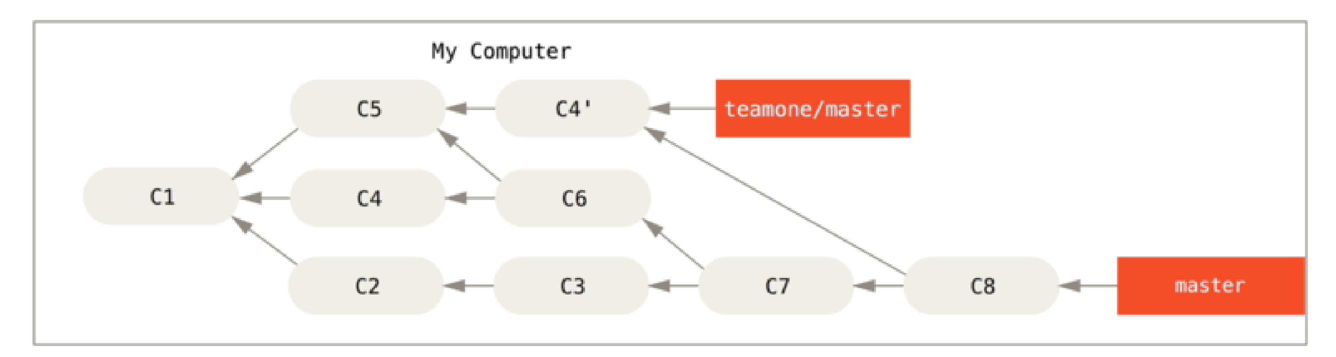


Figure 47. Vuelves a fusionar el mismo trabajo en una nueva fusión confirmada

Si ejecutas `git log` sobre un historial así, verás dos confirmaciones hechas por el mismo autor y con la misma fecha y mensaje, lo cual será confuso. Es más, si luego tu envías (push) ese registro de vuelta al servidor, vas a introducir todas esas confirmaciones reorganizadas en el servidor central. Lo que puede confundir aún más a la gente. Era más seguro asumir que el otro desarrollador no quería que `C4` y `C6` estuviesen en el historial; por ello había reorganizado su trabajo de esa manera.

Reorganizar una Reorganización

Si te encuentras en una situación como esta, Git tiene algunos trucos que pueden ayudarte. Si alguien de tu equipo sobreescribe cambios en los que se basaba tu trabajo, tu reto es descubrir qué han sobreescrito y qué te pertenece.

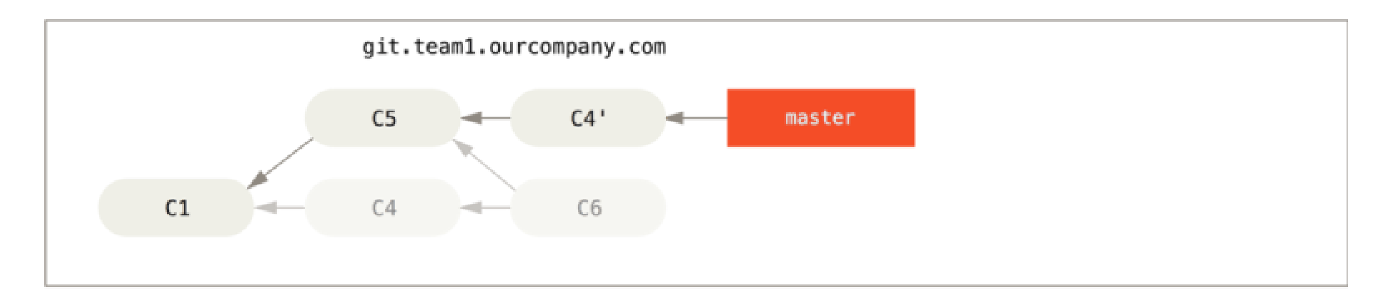
Además de la suma de control SHA-1, Git calcula una suma de control basada en el parche que introduce una confirmación. A esta se le conoce como “patch-id”.

Si te traes el trabajo que ha sido sobreescrito y lo reorganizas sobre las nuevas confirmaciones de tu compañero, es posible que Git pueda identificar qué parte correspondía específicamente a tu trabajo y aplicarla de vuelta en la rama nueva.

Por ejemplo, en el caso anterior, si en vez de hacer una fusión cuando estábamos en Alguien envio (push) confirmaciones (commits) reorganizadas, abandonando las confirmaciones en las que tu habías basado tu trabajo ejecutamos `git rebase teamone/master`, Git hará lo siguiente:

- Determinar el trabajo que es específico de nuestra rama (C2, C3, C4, C6, C7)
- Determinar cuáles no son fusiones confirmadas (C2, C3, C4)
- Determinar cuáles no han sido sobreescritas en la rama destino (solo C2 y C3, pues C4 corresponde al mismo parche que C4')
- Aplicar dichas confirmaciones encima de `teamone/master`

Así que en vez del resultado que vimos en Vuelves a fusionar el mismo trabajo en una nueva fusión confirmada, terminaremos con algo más parecido a Reorganizar encima de un trabajo sobreescrito reorganizado..



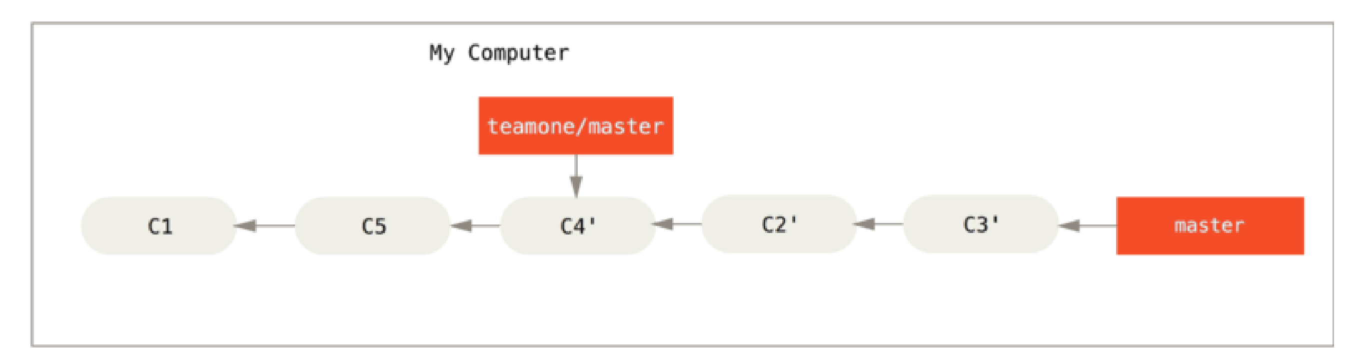


Figure 48. Reorganizar encima de un trabajo sobreescrito reorganizado.

Esto solo funciona si C4 y el C4' de tu compañero son parches muy similares. De lo contrario, la reorganización no será capaz de identificar que se trata de un duplicado y agregará otro parche similar a C4 (lo cual probablemente no podrá aplicarse limpiamente, pues los cambios ya estarían allí en algún lugar).

También puedes simplificar el proceso si ejecutas `git pull --rebase` en vez del tradicional `git pull`. O, en este caso, puedes hacerlo manualmente con un `git fetch` primero, seguido de un `git rebase teamone/master`.

Si sueles utilizar `git pull` y quieres que la opción `--rebase` esté activada por defecto, puedes asignar el valor de configuración `pull.rebase` haciendo algo como esto `git config --global pull.rebase true`.

Si consideras la reorganización como una manera de limpiar tu trabajo y tus confirmaciones antes de enviarlas (push), y si solo reorganizas confirmaciones (commits) que nunca han estado disponibles públicamente, no tendrás problemas. Si reorganizas (rebase) confirmaciones (commits) que ya estaban disponibles públicamente y la gente había basado su trabajo en ellas, entonces prepárate para tener problemas, frustrar a tu equipo y ser despreciado por tus compañeros.

Si tu compañero o tú ven que aun así es necesario hacerlo en algún momento, asegúrense que todos sepan que deben ejecutar `git pull --rebase` para intentar aliviar en lo posible la frustración.

Reorganizar vs. Fusionar

Ahora que has visto en acción la reorganización y la fusión, te preguntarás cuál es mejor. Antes de responder, repasemos un poco qué representa el historial.

Para algunos, el historial de confirmaciones de tu repositorio es **un registro de todo lo que ha pasado.** Un documento histórico, valioso por sí mismo y que no debería ser alterado. Desde este punto de vista, cambiar el historial de confirmaciones es casi como blasfemar; estarías *mintiendo* sobre lo que en verdad ocurrió. ¿Y qué pasa si hay una serie desastrosa de fusiones confirmadas? Nada. Así fue como ocurrió y el repositorio debería tener un registro de esto para la posteridad.

La otra forma de verlo es que el historial de confirmaciones es **la historia de cómo se hizo tu proyecto.** Tú no publicarías el primer borrador de tu novela, y el manual de cómo mantener tus programas también debe estar editado con mucho cuidado. Esta es el área que utiliza herramientas como `rebase` y `filter-branch` para contar la historia de la mejor manera para los futuros lectores.

Ahora, sobre qué es mejor si fusionar o reorganizar: verás que la respuesta no es tan sencilla. Git es una herramienta poderosa que te permite hacer muchas cosas con tu historial, y cada equipo y cada proyecto es diferente. Ahora que conoces cómo trabajan ambas herramientas, será cosa tuya decidir cuál de las dos es mejor para tu situación en particular.

Normalmente, la manera de sacar lo mejor de ambas es reorganizar tu trabajo local, que aun no has compartido, antes de enviarlo a algún lugar; pero nunca reorganizar nada que ya haya sido compartido.

Recapitulación

Hemos visto los procedimientos básicos de ramificación (branching) y fusión (merging) en Git. A estas alturas, te sentirás cómodo creando nuevas ramas (branch), saltando (checkout) entre ramas para trabajar y fusionando (merge) ramas entre ellas. También conocerás cómo compartir tus ramas enviándolas (push) a un servidor compartido, cómo trabajar colaborativamente en ramas compartidas, y cómo reorganizar (rebase) tus ramas antes de compartirlas. A continuación, hablaremos sobre lo que necesitas para tener tu propio servidor de hospedaje Git.

Git en el Servidor

En este punto, deberías ser capaz de realizar la mayoría de las tareas diarias para las cuales estarás usando Git. Sin embargo, para poder realizar cualquier colaboración en Git, necesitarás tener un repositorio remoto Git. Aunque técnicamente puedes enviar y recibir cambios desde repositorios de otros individuos, no se recomienda hacerlo porque, si no tienes cuidado, fácilmente podrías confundir en que es en lo que se está trabajando. Además, lo deseable es que tus colaboradores sean capaces de acceder al repositorio incluso si tu computadora no está en línea – muchas veces es útil tener un repositorio confiable en común. Por lo tanto, el método preferido para colaborar con otra persona es configurar un repositorio intermedio al cual ambos tengan acceso, y enviar (push) y recibir (pull) desde allí.

Poner en funcionamiento un servidor Git es un proceso bastante claro. Primero, eliges con qué protocolos ha de comunicarse tu servidor. La primera sección de este capítulo cubrirá los protocolos disponibles, así como los pros y los contras de cada uno. Las siguientes secciones explicarán algunas configuraciones comunes utilizando dichos protocolos y como poner a funcionar tu servidor con alguno de ellos. Finalmente, revisaremos algunas de las opciones hospedadas, si no te importa hospedar tu código en el servidor de alguien más y no quieres tomarte la molestia de configurar y mantener tu propio servidor.

Si no tienes interés en tener tu propio servidor, puedes saltarte hasta la última sección de este capítulo para ver algunas de las opciones para configurar una cuenta hospedada y seguir al siguiente capítulo, donde discutiremos los varios pormenores de trabajar en un ambiente de control de fuente distribuido.

Un repositorio remoto es generalmente un *repositorio básico* – un repositorio Git que no tiene directorio de trabajo. Dado que el repositorio es solamente utilizado como un punto de colaboración, no hay razón para tener una copia instantánea verificada en el disco; tan solo son datos Git. En los más simples términos, un repositorio básico es el contenido `.git` del directorio de tu proyecto y nada más.

Los Protocolos

Git puede usar cuatro protocolos principales para transferir datos: Local, HTTP, Secure Shell (SSH) y Git. Vamos a ver en qué consisten y las circunstancias en que querrás (o no) utilizar cada uno de ellos.

Local Protocol

El más básico es el *Protocolo Local*, donde el repositorio remoto es simplemente otra carpeta en el disco. Se utiliza habitualmente cuando todos los miembros del equipo tienen acceso a un mismo sistema de archivos, como por ejemplo un punto de montaje NFS, o en el caso menos frecuente de que todos se conectan al mismo computador. Aunque este último caso no es precisamente el ideal, ya que todas las instancias del repositorio estarían en la misma máquina; aumentando las posibilidades de una pérdida catastrófica.

Si dispones de un sistema de archivos compartido, podrás clonar (clone), enviar (push) y recibir (pull) a/desde repositorios locales basado en archivos. Para clonar un repositorio como estos, o para

añadirlo como remoto a un proyecto ya existente, usa el camino (path) del repositorio como su URL. Por ejemplo, para clonar un repositorio local, puedes usar algo como:

```
$ git clone /opt/git/project.git
```

O como:

```
$ git clone file:///opt/git/project.git
```

Git trabaja ligeramente distinto si indicas *file://* de forma explícita al comienzo de la URL. Si escribes simplemente el camino, Git intentará usar enlaces rígidos (hardlinks) o copiar directamente los archivos que necesita. Si escribes con el prefijo *file://*, Git lanza el proceso que usa habitualmente para transferir datos sobre una red; proceso que suele ser mucho menos eficiente. La única razón que puedes tener para indicar expresamente el prefijo *file://* puede ser el querer una copia limpia del repositorio, descartando referencias u objetos superfluos. Esto sucede normalmente, tras haberlo importado desde otro sistema de control de versiones o algo similar (ver Los entresijos internos de Git sobre tareas de mantenimiento). Habitualmente, usaremos el camino (path) normal por ser casi siempre más rápido.

Para añadir un repositorio local a un proyecto Git existente, puedes usar algo como:

```
$ git remote add local_proj /opt/git/project.git
```

Con lo que podrás enviar (push) y recibir (pull) desde dicho remoto exactamente de la misma forma a como lo harías a través de una red.

Ventajas

Las ventajas de los repositorios basados en carpetas y archivos, son su simplicidad y el aprovechamiento de los permisos preexistentes de acceso. Si tienes un sistema de archivo compartido que todo el equipo pueda usar, preparar un repositorio es muy sencillo. Simplemente pones el repositorio básico en algún lugar donde todos tengan acceso a él y ajustas los permisos de lectura/escritura según proceda, tal y como lo harías para preparar cualquier otra carpeta compartida. En la próxima sección, Configurando Git en un servidor, veremos cómo exportar un repositorio básico para conseguir esto.

Este camino es también util para recuperar rápidamente el contenido del repositorio de trabajo de alguna otra persona. Si tu y otra persona estáis trabajando en el mismo proyecto y ésta quiere mostrarte algo, el usar un comando tal como `git pull /home/john/project` suele ser más sencillo que el que esa persona te lo envie (push) a un servidor remoto y luego tú lo recojas (pull) desde allí.

Desventajas

La principal desventaja de los repositorios basados en carpetas y archivos es su dificultad de acceso desde distintas ubicaciones. Por ejemplo, si quieres enviar (push) desde tu portátil cuando estás en casa, primero tienes que montar el disco remoto; lo cual puede ser difícil y lento, en comparación

con un acceso basado en red.

Cabe destacar también que una carpeta compartida no es precisamente la opción más rápida. Un repositorio local es rápido solamente en aquellas ocasiones en que tienes un acceso rápido a él. Normalmente un repositorio sobre NFS es más lento que un repositorio SSH en el mismo servidor, asumiendo que las pruebas se hacen con Git sobre discos locales en ambos casos.

Protocolos HTTP

Git puede utilizar el protocolo HTTP de dos maneras. Antes de la versión 1.6.6 de Git, solo había una forma de utilizar el protocolo HTTP y normalmente en sólo lectura. Con la llegada de la versión 1.6.6 se introdujo un nuevo protocolo más inteligente que involucra a Git para negociar la transferencia de datos de una manera similar a como se hace con SSH. En los últimos años, este nuevo protocolo basado en HTTP se ha vuelto muy popular puesto que es más sencillo para el usuario y también más inteligente. Nos referiremos a la nueva versión como el HTTP "Inteligente" y llamaremos a la versión anterior el HTTP "tonto". Comenzaremos primero con el protocolo HTTP "Inteligente".

HTTP Inteligente

El protocolo HTTP "Inteligente" funciona de forma muy similar a los protocolos SSH y Git, pero se ejecutan sobre puertos estándar HTTP/S y pueden utilizar los diferentes mecanismos de autenticación HTTP. Esto significa que puede resultar más fácil para los usuarios, puesto que se pueden identificar mediante usuario y contraseña (usando la autenticación básica de HTTP) en lugar de usar claves SSH.

Es, probablemente, la forma más popular de usar Git ahora, puesto que puede configurarse para servir tanto acceso anónimo (como con el protocolo Git) y acceso autenticado para realizar envíos (push), con cifrado similar a como se hace con SSH. En lugar de tener diferentes URL para cada cosa, se puede tener una única URL para todo. Si intentamos subir cambios (push) al repositorio nos pedirá usuario y contraseña, y para accesos de lectura se puede permitir el acceso anónimo o requerir también usuario.

De hecho, para servicios como GitHub, la URL que usamos para ver el repositorio en la web (por ejemplo, "https://github.com/schacon/simplegit[]") es la misma que usaríamos para clonar y, si tenemos permisos, para enviar cambios.

HTTP Tonto

Si el servidor no dispone del protocolo HTTP "Inteligente", el cliente de Git intentará con el protocolo clásico HTTP que podemos llamar HTTP "Tonto". Este protocolo espera obtener el repositorio Git a través de un servidor web como si accediera a archivos normales. Lo bonito de este protocolo es la simplicidad para configurarlo. Básicamente, todo lo que tenemos que hacer es poner el repositorio Git bajo el directorio raíz de documentos HTTP y especificar un punto de enganche (hook) de `post-update` (véase Puntos de enganche en Git). Desde este momento, cualquiera con acceso al servidor web donde se publique el repositorio podrá también clonarlo. Para permitir acceso lectura con HTTP, debes hacer algo similar a lo siguiente:

```
$ cd /var/www/htdocs/
$ git clone --bare /path/to/git_project gitproject.git
$ cd gitproject.git
$ mv hooks/post-update.sample hooks/post-update
$ chmod a+x hooks/post-update
```

Y esto es todo. El punto de enganche `post-update` que trae Git de manera predeterminada ejecuta el comando adecuado (`git update-server-info`) para hacer que las operaciones de clonado o recuperación (fetch) funcionen de forma adecuada. Este comando se ejecuta cuando se envían cambios (push) al repositorio (mediante SSH, por ejemplo); luego, otras personas pueden clonar mediante algo como

```
$ git clone https://example.com/gitproject.git
```

En este caso concreto, hemos utilizado la carpeta `/var/www/htdocs`, que es el habitual en configuraciones Apache, pero se puede usar cualquier servidor web estático. Basta con que se ponga el repositorio básico (bare) en la carpeta correspondiente. Los datos de Git son servidos como archivos estáticos simples (véase Los entresijos internos de Git para saber exactamente cómo se sirven).

Por lo general tendremos que elegir servirlos en lectura/escritura con el servidor HTTP "Inteligente" o en solo lectura con el servidor "tonto". Mezclar ambos servicios no es habitual.

Ventajas

Nos centraremos en las ventajas de la versión "Inteligente" del protocolo HTTP.

La simplicidad de tener una única URL para todos los tipos de acceso y que el servidor pida autenticación solo cuando se necesite hace las cosas muy fáciles para el usuario final. Permitiendo autenticar mediante usuario y contraseña es también una ventaja sobre SSH, ya que los usuarios no tendrán que generar sus claves SSH y subir la pública al servidor antes de comenzar a usarlo. Esta es la principal ventaja para los usuarios menos especializados, o para los usuarios de sistemas donde el SSH no se suele usar. También es un protocolo muy rápido y eficiente, como sucede con el SSH.

También se pueden servir los repositorios en sólo lectura con HTTPS, lo que significa que se puede cifrar la transferencia de datos; incluso se puede identificar a los clientes haciéndoles usar certificados convenientemente firmados.

Otra cosa interesante es que los protocolos HTTP/S son los más ampliamente utilizados, de forma que los cortafuegos corporativos suelen permitir el tráfico a través de esos puertos.

Inconvenientes

Git sobre HTTP/S puede ser un poco más complejo de configurar comparado con el SSH en algunos sitios. En otros casos, se adivina poca ventaja sobre el uso de otros protocolos.

Si utilizamos HTTP para envíos autenticados, proporcionar nuestras credenciales cada vez que se

hace puede resultar más complicado que usar claves SSH. Hay, sin embargo, diversas utilidades de cacheo de credenciales, como Keychain en OSX o Credential Manager en Windows; haciendo esto menos incómodo. Lee Almacenamiento de credenciales para ver cómo configurar el cacheo seguro de contraseñas HTTP en tu sistema.

El Procotolo SSH

SSH es un protocolo muy habitual para alojar repositorios Git en hostings privados. Esto es así porque el acceso SSH viene habilitado de forma predeterminada en la mayoría de los servidores, y si no es así, es fácil de habilitarlo. Además, SSH es un protocolo de red autenticado, y es sencillo de utilizar.

Para clonar un repositorio a través de SSH, puedes indicar una URL ssh:// tal como:

```
$ git clone ssh://user@server/project.git
```

También puedes usar la sintaxis estilo scp del protocolo SSH:

```
$ git clone user@server:project.git
```

Pudiendo asimismo prescindir del usuario; en cuyo caso Git asume el usuario con el que estés conectado en ese momento.

Ventajas

El uso de SSH tiene múltiples ventajas. En primer lugar, SSH es relativamente fácil de configurar: los demonios (daemons) SSH son de uso común, muchos administradores de red tienen experiencia con ellos y muchas distribuciones del SO los traen predefinidos o tienen herramientas para gestionarlos. Además, el acceso a través de SSH es seguro, estando todas las transferencias encriptadas y autentificadas. Y, por último, al igual que los protocolos HTTP/S, Git y Local, SSH es eficiente, comprimiendo los datos lo más posible antes de transferirlos.

Desventajas

El aspecto negativo de SSH es su imposibilidad para dar acceso anónimo al repositorio. Todos han de tener configurado un acceso SSH al servidor, incluso aunque sea con permisos de solo lectura; lo que no lo hace recomendable para soportar proyectos abiertos. Si lo usas únicamente dentro de tu red corporativa, posiblemente sea SSH el único protocolo que tengas que emplear. Pero si quieres también habilitar accesos anónimos de solo lectura, tendrás que reservar SSH para tus envios (push) y habilitar algún otro protocolo para las recuperaciones (pull) de los demás.

El protocolo Git

El protocolo Git es un demonio (daemon) especial, que viene incorporado con Git. Escucha por un puerto dedicado (9418), y nos da un servicio similar al del protocolo SSH; pero sin ningún tipo de autentificación. Para que un repositorio pueda exponerse a través del protocolo Git, tienes que crear en él un archivo *git-daemon-export-ok*; sin este archivo, el demonio no hará disponible el

repositorio. Pero, aparte de esto, no hay ninguna otra medida de seguridad. O el repositorio está disponible para que cualquiera lo pueda clonar, o no lo está. Lo cual significa que, normalmente, no se podrá enviar (push) a través de este protocolo. Aunque realmente si que puedes habilitar el envio, si lo haces, dada la total falta de ningún mecanismo de autentificación, cualquiera que encuentre la URL a tu proyecto en Internet, podrá enviar (push) contenidos a él. Ni que decir tiene que esto solo lo necesitarás en contadas ocasiones.

Ventajas

El protocolo Git es el más rápido de todos los disponibles. Si has de servir mucho tráfico de un proyecto público o servir un proyecto muy grande, que no requiera autentificación para leer de él, un demonio Git es la respuesta. Utiliza los mismos mecanismos de transmisión de datos que el protocolo SSH, pero sin la sobrecarga de la encriptación ni de la autentificación.

Desventajas

La pega del protocolo Git, es su falta de autentificación. No es recomendable tenerlo como único protocolo de acceso a tus proyectos. Habitualmente, lo combinarás con un acceso SSH o HTTPS para los pocos desarrolladores con acceso de escritura que envíen (push) material, dejando el protocolo *git://* para los accesos solo-lectura del resto de personas.

Por otro lado, necesita activar su propio demonio, y necesita configurar *xinetd* o similar, lo cual no suele estar siempre disponible en el sistema donde estés trabajando. Requiere además abrir expresamente acceso al puerto 9418 en el cortafuegos, ya que estará cerrado en la mayoría de los cortafuegos corporativos.

Configurando Git en un servidor

Ahora vamos a cubrir la creación de un servicio de Git ejecutando estos protocolos en su propio servidor.

NOTE Aquí demostraremos los comandos y pasos necesarios para hacer las instalaciones básicas simplificadas en un servidor basado en Linux, aunque también es posible ejecutar estos servicios en los servidores Mac o Windows. Configurar un servidor de producción dentro de tu infraestructura sin duda supondrá diferencias en las medidas de seguridad o de las herramientas del sistema operativo, pero se espera que esto le de la idea general de lo que el proceso involucra.

Para configurar por primera vez un servidor de Git, hay que exportar un repositorio existente en un nuevo repositorio vacío - un repositorio que no contiene un directorio de trabajo. Esto es generalmente fácil de hacer. Para clonar el repositorio con el fin de crear un nuevo repositorio vacío, se ejecuta el comando clone con la opción `--bare`. Por convención, los directorios del repositorio vacío terminan en .git , así:

```
$ git clone --bare my_project my_project.git
Cloning into bare repository 'my_project.git'...
done.
```

Deberías tener ahora una copia de los datos del directorio Git en tu directorio `my_project.git`. Esto es más o menos equivalente a algo así:

```
$ cp -Rf my_project/.git my_project.git
```

Hay un par de pequeñas diferencias en el archivo de configuración; pero para tú propósito, esto es casi la misma cosa. Toma el repositorio Git en sí mismo, sin un directorio de trabajo, y crea un directorio específicamente para él solo.

Colocando un Repositorio Vacío en un Servidor

Ahora que tienes una copia vacía de tú repositorio, todo lo que necesitas hacer es ponerlo en un servidor y establecer sus protocolos. Digamos que has configurado un servidor llamado `git.example.com` que tiene acceso a SSH, y quieres almacenar todos tus repositorios Git bajo el directorio / opt` / git`. Suponiendo que existe `/ opt / git` en ese servidor, puedes configurar tu nuevo repositorio copiando tu repositorio vacío a:

```
$ scp -r my_project.git user@git.example.com:/opt/git
```

En este punto, otros usuarios que con acceso SSH al mismo servidor que tiene permisos de lectura-acceso al directorio `/ opt / git` pueden clonar tu repositorio mediante el comando

```
$ git clone user@git.example.com:/opt/git/my_project.git
```

Si un usuario accede por medio de SSH a un servidor y tiene permisos de escritura en el directorio `git my_project.git` / opt / /, automáticamente también tendrán acceso push.

Git automáticamente agrega permisos de grupo para la escritura en un repositorio apropiadamente si se ejecuta el comando `git init` con la opción` --shared`.

```
$ ssh user@git.example.com
$ cd /opt/git/my_project.git
$ git init --bare --shared
```

Puedes ver lo fácil que es tomar un repositorio Git, crear una versión vacía, y colocarlo en un servidor al que tú y tus colaboradores tienen acceso SSH. Ahora estan listos para colaborar en el mismo proyecto.

Es importante tener en cuenta que esto es literalmente todo lo que necesitas hacer para ejecutar un útil servidor Git al cual varias personas tendrán acceso - sólo tiene que añadir cuentas con acceso SSH a un servidor, y subir un repositorio vacío en alguna parte a la que todos los usuarios puedan leer y escribir. Estás listo para trabajar. Nada más es necesario.

En las próximas secciones, verás cómo ampliar a configuraciones más sofisticadas. Esta sección incluirá no tener que crear cuentas para cada usuario, añadiendo permisos de lectura pública a los

repositorios, la creación de interfaces de usuario web y más. Sin embargo, ten en cuenta que para colaborar con un par de personas en un proyecto privado, todo_lo_que_necesitas_es un servidor SSH y un repositorio vacío.

Pequeñas configuraciones

Si tienes un pequeño equipo o acabas de probar Git en tu organización y tienes sólo unos pocos desarrolladores, las cosas pueden ser simples para ti. Uno de los aspectos más complicados de configurar un servidor Git es la gestión de usuarios. Si quieres que algunos repositorios sean de sólo lectura para ciertos usuarios y lectura / escritura para los demás, el acceso y los permisos pueden ser un poco más difíciles de organizar.

Acceso SSH

Si tienes un servidor al que todos los desarrolladores ya tienen acceso SSH, es generalmente más fácil de configurar el primer repositorio allí, porque no hay que hacer casi ningún trabajo (como ya vimos en la sección anterior). Si quieres permisos de acceso más complejas en tus repositorios, puedes manejarlos con los permisos del sistema de archivos normales del sistema operativo donde tu servidor se ejecuta.

Si deseas colocar los repositorios en un servidor que no tiene cuentas para todo el mundo en su equipo para el que deseas tener acceso de escritura, debes configurar el acceso SSH para ellos. Suponiendo que tienes un servidor con el que hacer esto, ya tiene un servidor SSH instalado, y así es como estás accediendo al servidor.

Hay algunas maneras con las cuales puedes dar acceso a todo tu equipo. La primera es la creación de cuentas para todo el mundo, que es sencillo, pero puede ser engorroso. Podrías no desear ejecutar `adduser` y establecer contraseñas temporales para cada usuario.

Un segundo método consiste en crear un solo usuario *git* en la máquina, preguntar a cada usuario de quién se trata para otorgarle permisos de escritura para que te envíe una llave SSH pública, y agregar esa llave al archivo `~ / .ssh / authorized_keys` de tu nuevo usuario *git*. En ese momento, todo el mundo podrá acceder a esa máquina mediante el usuario *git*. Esto no afecta a los datos commit de ninguna manera - el usuario SSH con el que te conectas no puede modificar los commits que has registrado.

Otra manera de hacer que tu servidor SSH autentifique desde un servidor LDAP o desde alguna otra fuente de autentificación centralizada que pudieras tener ya configurada. Mientras que cada usuario sea capaz de tener acceso shell a la máquina, cualquier mecanismo de autentificación SSH que se te ocurra debería de funcionar.

Generando tu clave pública SSH

Tal y como se ha comentado, muchos servidores Git utilizan la autentificación a través de claves públicas SSH. Y, para ello, cada usuario del sistema ha de generarse una, si es que no la tiene ya. El proceso para hacerlo es similar en casi cualquier sistema operativo. Ante todo, asegurarte que no tengas ya una clave. Por defecto, las claves de cualquier usuario SSH se guardan en la carpeta `~/.ssh` de dicho usuario. Puedes verificar si tienes ya unas claves, simplemente situándote sobre dicha carpeta y viendo su contenido:

```
$ cd ~/.ssh
$ ls
authorized_keys2  id_dsa       known_hosts
config            id_dsa.pub
```

Has de buscar un par de archivos con nombres tales como *algo* y *algo.pub*; siendo ese "algo" normalmente *id_dsa* o *id_rsa*. El archivo terminado en *.pub* es tu clave pública, y el otro archivo es tu clave privada. Si no tienes esos archivos (o no tienes ni siquiera la carpeta *.ssh*), has de crearlos; utilizando un programa llamado *ssh-keygen*, que viene incluido en el paquete SSH de los sistemas Linux/Mac o en el paquete MSysGit en los sistemas Windows:

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/schacon/.ssh/id_rsa):
Created directory '/home/schacon/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/schacon/.ssh/id_rsa.
Your public key has been saved in /home/schacon/.ssh/id_rsa.pub.
The key fingerprint is:
d0:82:24:8e:d7:f1:bb:9b:33:53:96:93:49:da:9b:e3 schacon@mylaptop.local
```

Como se ve, este comando primero solicita confirmación de dónde van a a guardarse las claves (*.ssh/id_rsa*), y luego solicita, dos veces, una contraseña (passphrase), contraseña que puedes dejar en blanco si no deseas tener que teclearla cada vez que uses la clave.

Tras generarla, cada usuario ha de encargarse de enviar su clave pública a quienquiera que administre el servidor Git (en el caso de que éste esté configurado con SSH y así lo requiera). Esto se puede realizar simplemente copiando los contenidos del archivo terminado en *.pub* y enviándoselos por correo electrónico. La clave pública será una serie de números, letras y signos, algo así como esto:

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAklOUpkDHrfHY17SbrmTIpNLTGK9Tjom/BWDSU
GPl+nafzlHDTYW7hdI4yZ5ew18JH4JW9jbhUFrviQzM7xlELEVf4h9lFX5QVkbPppSwg0cda3
Pbv7kOdJ/MTyBlWXFCR+HAo3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSlVK/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUFljQJKprrX88XypNDvjYNby6vw/Pb0rwert/En
mZ+AW4OZPnTPI89ZPmVMLuayrD2cE86Z/il8b+gw3r3+1nKatmIkjn2so1d01QraTlMqVSsbx
NrRFi9wrf+M7Q== schacon@mylaptop.local
```

Para más detalles sobre cómo crear unas claves SSH en variados sistemas operativos, consultar la correspondiente guía en GitHub: https://help.github.com/articles/generating-ssh-keys.

Configurando el servidor

Vamos a avanzar en los ajustes de los accesos SSH en el lado del servidor. En este ejemplo, usarás el

método de las `authorized_keys` (claves autorizadas) para autentificar a tus usuarios. Se asume que tienes un servidor en marcha, con una distribución estándar de Linux, tal como Ubuntu. Comienzas creando un usuario *git* y una carpeta `.ssh` para él.

```
$ sudo adduser git
$ su git
$ cd
$ mkdir .ssh && chmod 700 .ssh
$ touch .ssh/authorized_keys && chmod 600 .ssh/authorized_keys
```

Y a continuación añades las claves públicas de los desarrolladores al archivo `authorized_keys` del usuario `git` que has creado. Suponiendo que hayas recibido las claves por correo electrónico y que las has guardado en archivos temporales. Y recordando que las claves públicas son algo así como:

```
$ cat /tmp/id_rsa.john.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCB007n/ww+ouN4gSLKssMxXnBOvf9LGt4L
ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4k
Yjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4oxOj6H0rfIF1kKI9MAQLMdpGW1GYEIgS9Ez
Sdfd8AcCIicTDWbqLAcU4UpkaX8KyGlLwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myiv
O7TCUSBdLQlgMVOFq1I2uPWQOkOWQAHukEOmfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPq
dAv8JggJICUvax2T9va5 gsg-keypair
```

No tienes más que añadirlas al archivo `authorized_keys` dentro del directorio `.ssh`:

```
$ cat /tmp/id_rsa.john.pub >> ~/.ssh/authorized_keys
$ cat /tmp/id_rsa.josie.pub >> ~/.ssh/authorized_keys
$ cat /tmp/id_rsa.jessica.pub >> ~/.ssh/authorized_keys
```

Tras esto, puedes preparar un repositorio básico vacio para ellos, usando el comando `git init` con la opción `--bare` para inicializar el repositorio sin carpeta de trabajo:

```
$ cd /opt/git
$ mkdir project.git
$ cd project.git
$ git init --bare
Initialized empty Git repository in /opt/git/project.git/
```

Y John, Josie o Jessica podrán enviar (push) la primera versión de su proyecto a dicho repositorio, añadiéndolo como remoto y enviando (push) una rama (branch). Cabe indicar que alguien tendrá que iniciar sesión en la máquina y crear un repositorio básico, cada vez que se desee añadir un nuevo proyecto. Suponiendo, por ejemplo, que se llame `gitserver` el servidor donde has puesto el usuario `git` y los repositorios; que dicho servidor es interno a vuestra red y que está asignado el nombre `gitserver` en vuestro DNS. Podrás utilizar comandos tales como (suponiendo que `myproject` es un proyecto ya creado con algunos archivos):

```
# on Johns computer
$ cd myproject
$ git init
$ git add .
$ git commit -m 'initial commit'
$ git remote add origin git@gitserver:/opt/git/project.git
$ git push origin master
```

Tras lo cual, otros podrán clonarlo y enviar cambios de vuelta:

```
$ git clone git@gitserver:/opt/git/project.git
$ cd project
$ vim README
$ git commit -am 'fix for the README file'
$ git push origin master
```

Con este método, puedes preparar rápidamente un servidor Git con acceso de lectura/escritura para un grupo de desarrolladores.

Observa que todos esos usuarios pueden también entrar en el servidor obteniendo un intérprete de comandos con el usuario `git`. Si quieres restringirlo, tendrás que cambiar el intérprete (shell) en el archivo `passwd`.

Para una mayor protección, puedes restringir fácilmente el usuario `git` a realizar solamente actividades relacionadas con Git, utilizando un shell limitado llamado `git-shell`, que viene incluido en Git. Si lo configuras como el shell de inicio de sesión de tu usuario `git`, dicho usuario no tendrá acceso al shell normal del servidor. Para especificar el `git-shell` en lugar de bash o de csh como el shell de inicio de sesión de un usuario, has de editar el archivo */etc/passwd*:

```
$ cat /etc/shells   # mirar si `git-shell` ya está aquí.  Si no...
$ which git-shell   # buscar `git-shell` en nuestro sistema
$ sudo vim /etc/shells  # y añadirlo al final de este archivo con el camino (path)
completo
```

Ahora ya puedes cambiar la shell del usuario utilizando `chsh <username>`:

```
$ sudo chsh git  # poner aquí la nueva shell, normalmente será: /usr/bin/git-shell
```

De esta forma dejamos al usuario *git* limitado a utilizar la conexión SSH solamente para enviar (push) y recibir (pull) repositorios, sin posibilidad de iniciar una sesión normal en el servidor. Si pruebas a hacerlo, recibirás un rechazo de inicio de sesión:

```
$ ssh git@gitserver
fatal: Interactive git shell is not enabled.
hint: ~/git-shell-commands should exist and have read and execute access.
Connection to gitserver closed.
```

Los comandos remotos de Git funcionarán con normalidad, pero los usuarios no podrán obtener un intérprete de comandos del sistema. Tal como nos avisa, también se puede establecer un directorio llamado `git-shell-commands` en la cuenta del usuario `git` para personalizar un poco el git-shell. Por ejemplo, se puede restringir qué comandos de Git se aceptarán o se puede personalizar el mensaje que los usuarios verán si intentan abrir un intérprete de comandos con SSH.

Ejecutando `git help shell` veremos más información sobre cómo personalizar el shell.

El demonio Git

Ahora vamos a configurar un demonio sirviendo repositorios mediante el protocolo "Git". Es la forma mas común para dar acceso anónimo pero rápido a los repositorios. Recuerda que puesto que es un acceso no autentificado, todo lo que sirvas mediante este protocolo será público en la red.

Si activas el protocolo en un servidor más allá del cortafuegos, lo debes usar únicamente en proyectos que deban ser visibles a todo el mundo. Si el servidor está detrás de un cortafuegos, puedes usarlo en proyectos a los que un gran número de personas o de computadores (por ejemplo, servidores de integración continua o de compilación) tengan acceso de sólo lectura y no necesiten establecer una clave SSH para cada uno de ellos.

El protocolo Git es relativamente fácil de configurar. Básicamente, necesitas ejecutar el comando con la variante demonio (daemon):

```
git daemon --reuseaddr --base-path=/opt/git/ /opt/git/
```

El parámetro `--reuseaddr` permite al servidor reiniciarse sin esperar a que se liberen viejas conexiones; el parámetro `--base-path` permite a los usuarios clonar proyectos sin necesidad de indicar su camino completo; y el camino indicado al final del comando mostrará al demonio Git dónde buscar los repositorios a exportar. Si tienes un cortafuegos activo, necesitarás abrir el puerto 9418 para la máquina donde estás configurando el demonio Git.

Este proceso se puede demonizar de diferentes maneras, dependiendo del sistema operativo con el que trabajas. En una máquina Ubuntu, puedes usar un script de arranque. Poniendo en el siguiente archivo:

```
/etc/event.d/local-git-daemon
```

un script tal como:

```
start on startup
stop on shutdown
exec /usr/bin/git daemon \
    --user=git --group=git \
    --reuseaddr \
    --base-path=/opt/git/ \
    /opt/git/
respawn
```

Por razones de seguridad, es recomendable lanzar este demonio con un usuario que tenga únicamente permisos de lectura en los repositorios (Lo puedes hacer creando un nuevo usuario *git-ro* y lanzando el demonio con él). Para simplificar, en estos ejemplos vamos a lanzar el demonio Git bajo el mismo usuario `git` que se usa con `git-shell`.

Tras reiniciar tu máquina, el demonio Git arrancará automáticamente y se reiniciará cuando se caiga. Para arrancarlo sin necesidad de reiniciar la máquina, puedes utilizar el comando:

```
initctl start local-git-daemon
```

En otros sistemas operativos, puedes utilizar `xinetd`, un script en el sistema `sysvinit`, o alguna otra manera (siempre y cuando demonizes el comando y puedas monitorizarlo).

A continuación, has de indicar a Git a cuales de tus repositorios ha de permitir acceso sin autentificar. Lo puedes hacer creando en cada repositorio un archivo llamado `git-daemon-export-ok`.

```
$ cd /path/to/project.git
$ touch git-daemon-export-ok
```

La presencia de este archivo dice a Git que este proyecto se puede servir sin problema sin necesidad de autentificación de usuarios.

HTTP Inteligente

Ahora ya tenemos acceso autentificado mediante SSH y anónimo mediante `git://`, pero hay también otro protocolo que permite tener ambos accesos a la vez. Configurar HTTP inteligente consiste, básicamente, en activar en el servidor web un script CGI que viene con Git, llamado `git-http-backend`. Este CGI leerá la ruta y las cabeceras enviadas por los comandos `git fetch` o `git push` a una URL de HTTP y determinará si el cliente puede comunicar con HTTP (lo que será cierto para cualquier cliente a partir de la versión 1.6.6). Si el CGI comprueba que el cliente es inteligente, comunicará inteligentemente con él; en otro caso pasará a usar el comportamiento tonto (es decir, es compatible con versiones más antiguas del cliente).

Revisemos una configuración básica. Pondremos Apache como servidor de CGI. Si no tienes Apache configurado, lo puedes instalar en un Linux con un comando similar a este:

```
$ sudo apt-get install apache2 apache2-utils
$ a2enmod cgi alias env
```

Esto además activa los módulos mod_cgi, mod_alias, y mod_env, que van a hacer falta para que todo esto funcione.

A continuación tenemos que añadir algunas cosas a la configuración de Apache para que se utilice git-http-backend para cualquier cosa que haya bajo la carpeta virtual /git.

```
SetEnv GIT_PROJECT_ROOT /opt/git
SetEnv GIT_HTTP_EXPORT_ALL
ScriptAlias /git/ /usr/libexec/git-core/git-http-backend/
```

Si dejas sin definir la variable de entorno GIT_HTTP_EXPORT_ALL, Git solo servirá a los clientes anónimos aquellos repositorios que contengan el archivo daemon-export-ok, igual que hace el demonio Git.

Ahora tienes que decirle a Apache que acepte peticiones en esta ruta con algo similar a esto:

```
<Directory "/usr/lib/git-core*">
   Options ExecCGI Indexes
   Order allow,deny
   Allow from all
   Require all granted
</Directory>
```

Finalmente, si quieres que los clientes autentificados tengan acceso de escritura, tendrás que crear un bloque Auth similar a este:

```
<LocationMatch "^/git/.*/git-receive-pack$">
    AuthType Basic
    AuthName "Git Access"
    AuthUserFile /opt/git/.htpasswd
    Require valid-user
</LocationMatch>
```

Esto requiere que hagas un archivo .htaccess que contenga las contraseñas cifradas de todos los usuarios válidos. Por ejemplo, para añadir el usuario “schacon” a este archivo:

```
$ htdigest -c /opt/git/.htpasswd "Git Access" schacon
```

Hay un montón de maneras de dar acceso autentificado a los usuarios con Apache, y tienes que elegir una. Esta es la forma más simple de hacerlo. Probablemente también te interese hacerlo todo con SSL para que todos los datos vayan cifrados.

No queremos profundizar en los detalles de la configuración de apache, ya que puedes tener diferentes necesidades de autentificación o querer utilizar un servidor diferente. La idea es que Git trae un CGI llamado `git-http-backend` que cuando es llamado, hace toda la negociación y envío o recepción de datos a través de HTTP. Por sí mismo no implementa autentificación de ningún tipo, pero puede controlarse desde el servidor web que lo utiliza. Puedes configurar esto en casi cualquier servidor web que pueda trabajar con CGI, el que más te guste.

NOTE	Para más información sobre cómo configurar Apache, mira la documentación: http://httpd.apache.org/docs/current/howto/auth.html

GitWeb

Ahora que ya tienes acceso básico de lectura/escritura y de solo-lectura a tu proyecto, puedes querer instalar un visualizador web. Git trae un script CGI, denominado GitWeb, que es el que usaremos para este propósito.

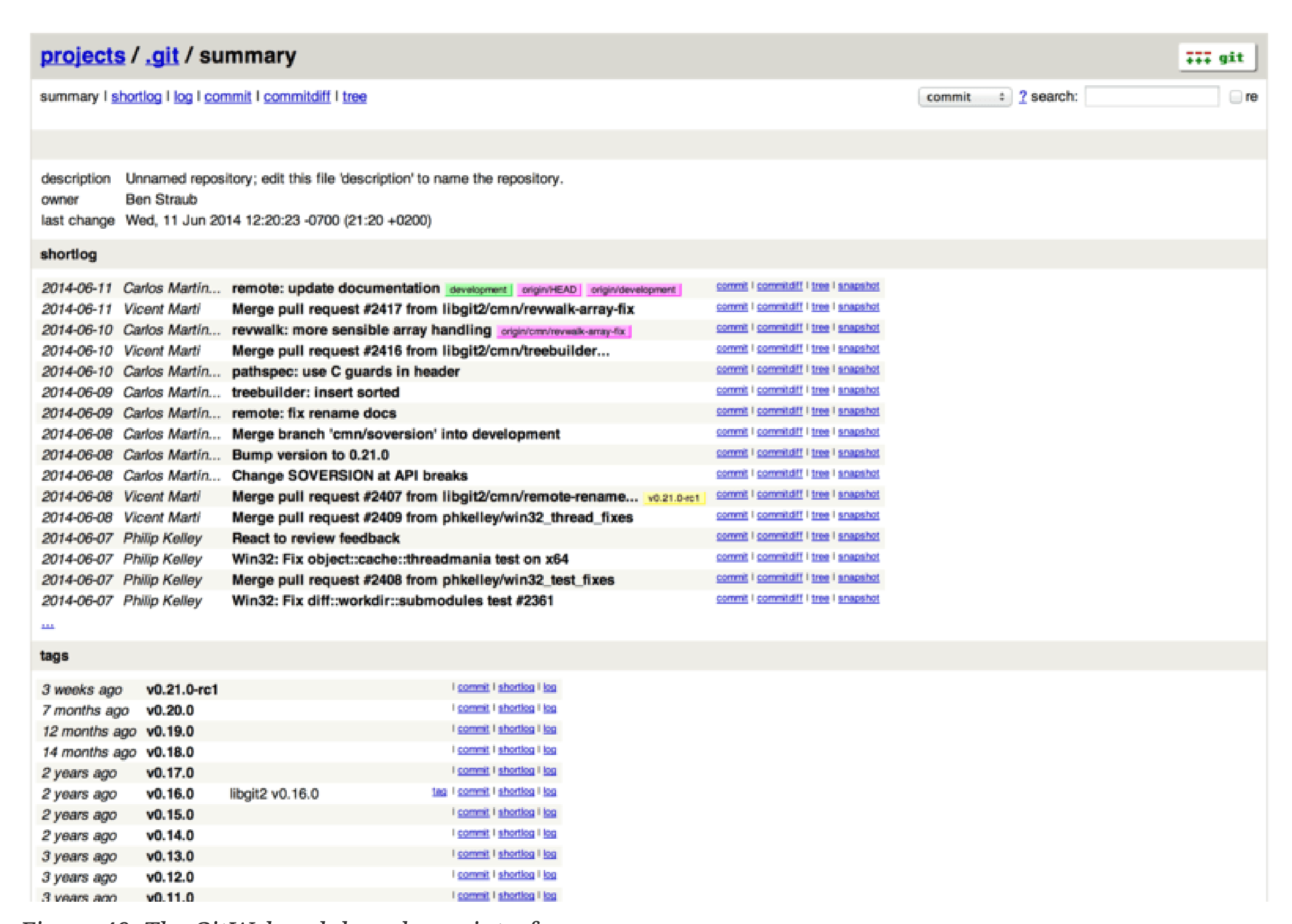


Figure 49. The GitWeb web-based user interface.

Si quieres comprobar cómo podría quedar GitWeb con tu proyecto, Git dispone de un comando para activar una instancia temporal, si en tu sistema tienes un servidor web ligero, como por ejemplo `lighttpd` o `webrick`. En las máquinas Linux, `lighttpd` suele estar habitualmente instalado, por lo que tan solo has de activarlo lanzando el comando `git instaweb`, estando en la carpeta de tu proyecto. Si tienes una máquina Mac, Leopard trae preinstalado Ruby, por lo que `webrick` puede ser tu mejor apuesta. Para instalar `instaweb` disponiendo de un controlador no-lighttpd, puedes lanzarlo con la opción `--httpd`.

```
$ git instaweb --httpd=webrick
[2009-02-21 10:02:21] INFO  WEBrick 1.3.1
[2009-02-21 10:02:21] INFO  ruby 1.8.6 (2008-03-03) [universal-darwin9.0]
```

Esto arranca un servidor HTTPD en el puerto 1234, y luego arranca un navegador que abre esa página. Es realmente sencillo. Cuando ya hayas terminado y quieras apagar el servidor, puedes lanzar el mismo comando con la opción `--stop`:

```
$ git instaweb --httpd=webrick --stop
```

Si quieres disponer permanentemente de un interfaz web para tu equipo o para un proyecto de código abierto que albergues, necesitarás ajustar el script CGI para ser servido por tu servidor web habitual. Algunas distribuciones Linux suelen incluir el paquete `gitweb`, y podrás instalarlo a través de las utilidades `apt` o `yum`; merece la pena probarlo en primer lugar. Enseguida vamos a revisar el proceso de instalar GitWeb manualmente. Primero, necesitas el código fuente de Git, que viene con GitWeb, para generar un script CGI personalizado:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/
$ make GITWEB_PROJECTROOT="/opt/git" prefix=/usr gitweb
    SUBDIR gitweb
    SUBDIR ../
make[2]: `GIT-VERSION-FILE' is up to date.
    GEN gitweb.cgi
    GEN static/gitweb.js
$ sudo cp -Rf gitweb /var/www/
```

Fíjate que es necesario indicar la ubicación donde se encuentran los repositorios Git, utilizando la variable *GITWEB_PROJECTROOT*. A continuación, tienes que preparar Apache para que utilice dicho script, Para ello, puedes añadir un VirtualHost:

```
<VirtualHost *:80>
    ServerName gitserver
    DocumentRoot /var/www/gitweb
    <Directory /var/www/gitweb>
        Options ExecCGI +FollowSymLinks +SymLinksIfOwnerMatch
        AllowOverride All
        order allow,deny
        Allow from all
        AddHandler cgi-script cgi
        DirectoryIndex gitweb.cgi
    </Directory>
</VirtualHost>
```

Recordar una vez más que GitWeb puede servirse desde cualquier servidor web con capacidades

CGI o Perl. Por lo que si prefieres utilizar algún otro, no debería ser difícil configurarlo. En este momento, deberías poder visitar `http://gitserver/` para ver tus repositorios online.

GitLab

GitWeb es muy simple. Si buscas un servidor Git más moderno, con todas las funciones, tienes algunas soluciones de código abierto que puedes utilizar en su lugar. Puesto que GitLab es una de las más populares, vamos a ver aquí cómo se instala y se usa, a modo de ejemplo. Es algo más complejo que GitWeb y requiere algo más de mantenimiento, pero es una opción con muchas más funciones.

Instalación

GitLab es una aplicación web con base de datos, por lo que su instalación es algo más complicada. Por suerte, es un proceso muy bien documentado y soportado.

Hay algunos métodos que puedes seguir para instalar GitLab. Para tener algo rápidamente, puedes descargar una máquina virtual o un instalador one-click desde https://bitnami.com/stack/gitlab, y modificar la configuración para tu caso particular. La pantalla de inicio de Bitnami (a la que se accede con alt-→); te dirá la dirección IP y el usuario y contraseña utilizados para instalar GitLab.

```
*** Welcome to the BitNami Gitlab Stack ***
*** Built using Ubuntu 12.04 - Kernel 3.2.0-53-virtual (tty2). ***

*** You can access the application at http://10.0.1.17 ***
*** The default username and password is 'user@example.com' and 'bitnami1'. ***
*** Please refer to http://wiki.bitnami.com/Virtual_Machines for details. ***

linux login: _
```

Figure 50. Página de login de la máquina virtual Bitnami.

Para las demás cosas, utiliza como guía los archivos readme de la edición Community de GitLab, que se pueden encontrar en https://gitlab.com/gitlab-org/gitlab-ce/tree/master. Aquí encontrarás ayuda para instalar Gitlab usando recetas Chef, una máquina virtual para Digital Ocean, y paquetes RPM y DEB (los cuales, en el momento de escribir esto, aun estaban en beta). También hay guías “no oficiales” para configurar GitLab en sistemas operativos o con bases de datos no estándar, un script de instalación completamente manual y otros muchos temas.

Administración

La interfaz de administración de GitLab se accede mediante la web. Simplemente abre en tu navegador la IP o el nombre de máquina donde has instalado Gitlab, y entra con el usuario administrador. El usuario predeterminado es `admin@local.host`, con la contraseña `5iveL!fe` (que te pedirá cambiar cuando entres por primera vez). Una vez dentro, pulsa en el icono “Admin area” del menú superior derecho.

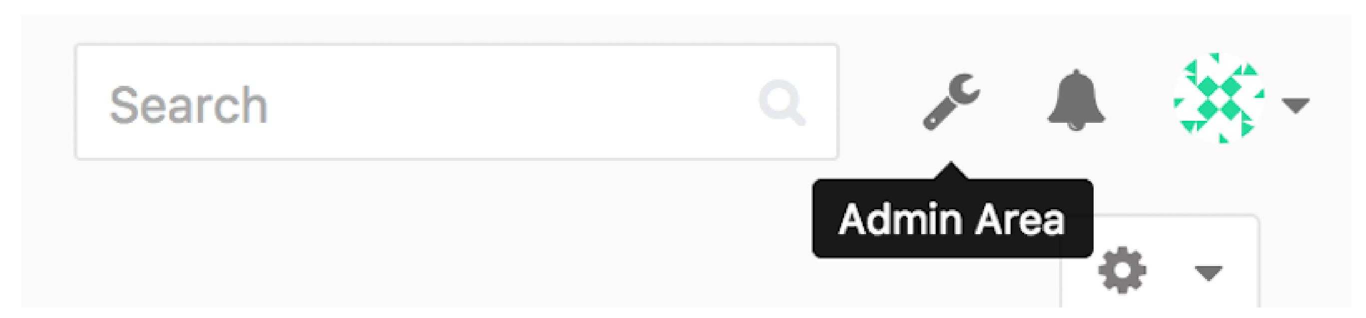


Figure 51. El icono “Admin area” del menú de GitLab.

Usuarios

Los usuarios en Gitlab son las cuentas que abre la gente. Las cuentas de usuario no tienen ninguna complicación: viene a ser una colección de información personal unida a la información de login. Cada cuenta tiene un **espacio de nombres** (namespace) que es una agrupación lógica de los proyectos que pertenecen al usuario. De este modo, si el usuario `jane` tiene un proyecto llamado `project`, la URL de ese proyecto sería http://server/jane/project.

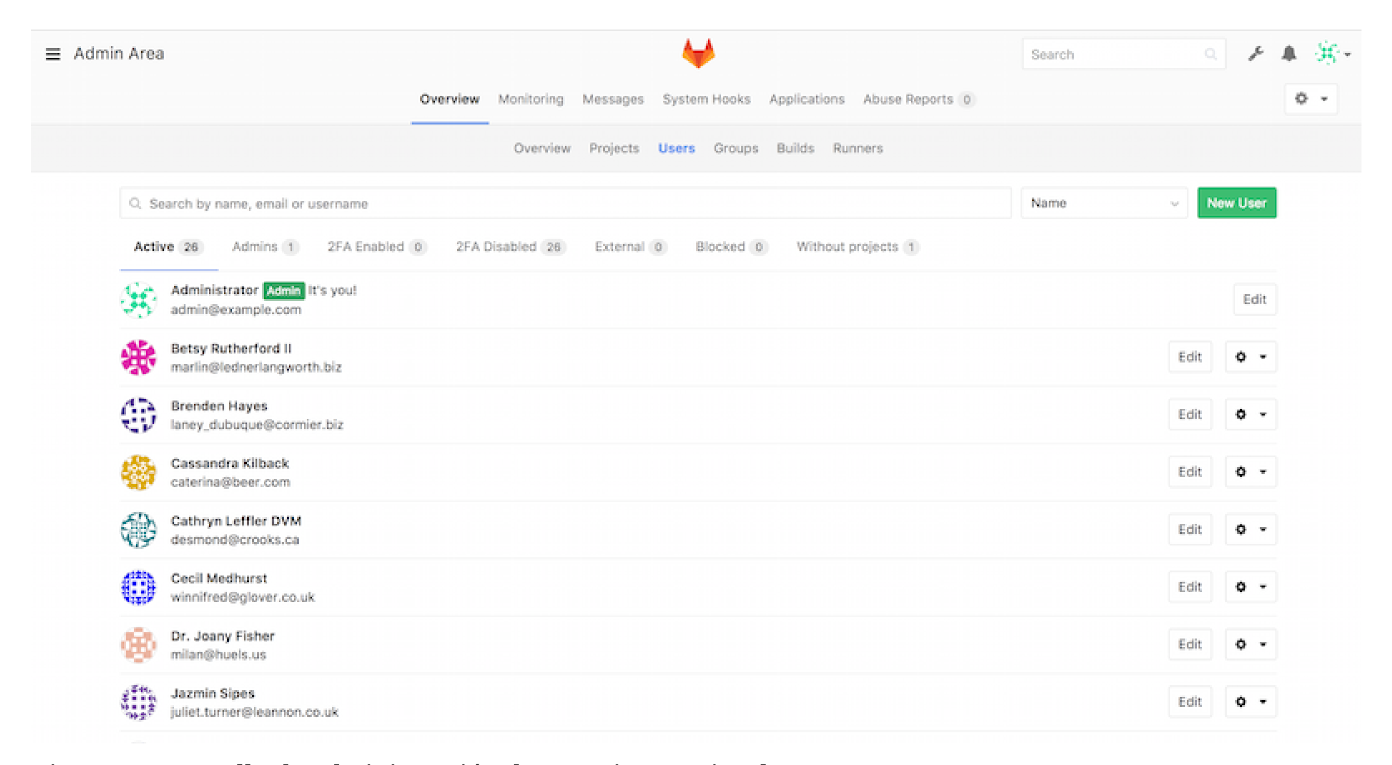


Figure 52. Pantalla de administración de usuarios en GitLab.

Tenemos dos formas de borrar usuarios. “Bloquear” un usuario evita que el usuario entre en Gitlab, pero los datos de su espacio de nombres se conservan, y los commits realizados por el usuario seguirán a su nombre y relacionados con su perfil.

“Destruir” un usuario, por su parte, borra completamente al usuario de la base de datos y el sistema

de archivos. Todos los proyectos y datos de su espacio de nombres se perderá, así como cualquier grupo que le pertenezca. Esto es, por supuesto, la acción más permanente y destructiva, y casi nunca se usa.

Grupos

Un grupo de GitLab es un conjunto de proyectos, junto con los datos acerca de los usuarios que tienen acceso. Cada grupo tiene también un espacio de nombres específico (al igual que los usuarios). Por ejemplo, si el grupo `formacion` tuviese un proyecto `materiales` su URL sería: http://server/formacion/materiales.

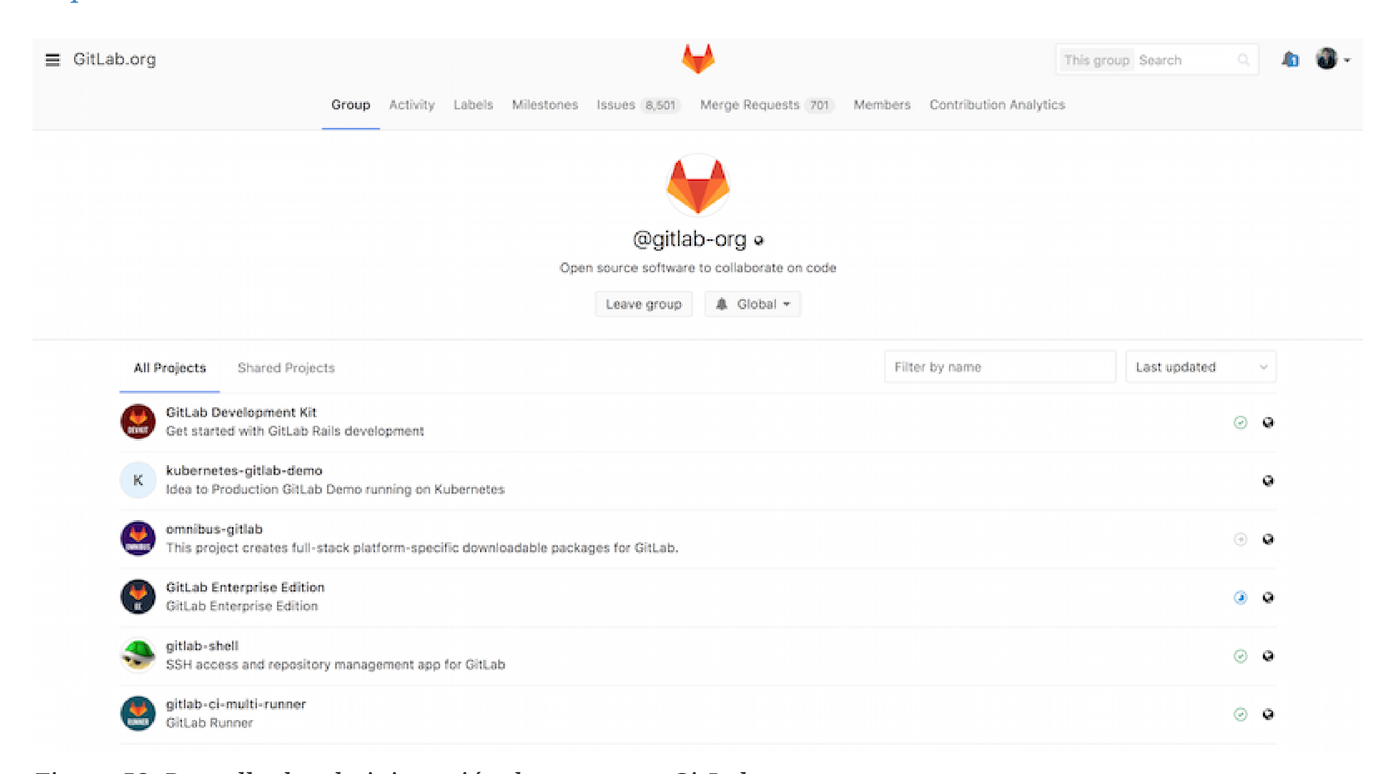


Figure 53. Pantalla de administración de grupos en GitLab.

Cada grupo se asocia con un conjunto de usuarios, y cada usuario tiene un nivel de permisos sobre los proyectos así como el propio grupo. Estos permisos van desde el de “Invitado” (que solo permite manejar incidencias y chat) hasta el de “Propietario” (con control absoluto del grupo, sus miembros y sus proyectos). Los tipos de permisos son muy numerosos para detallarlos aquí, pero en la ayuda de la pantalla de administración de GitLab la encontraremos fácilmente.

Proyectos

Un proyecto en GitLab corresponde con un repositorio Git. Cada proyecto pertenece a un espacio de nombres, bien sea de usuario o de grupo. Si el proyecto pertenece a un usuario, el propietario del mismo tendrá control directo sobre quién tiene acceso al proyecto; si el proyecto pertenece a un grupo, los permisos de acceso por parte de los usuarios estarán también determinados por los niveles de acceso de los miembros del grupo.

Cada proyecto tiene también un nivel de visibilidad, que controla quién tiene acceso de lectura a las páginas del proyecto y al propio repositorio. Si un proyecto es *Privado*, el propietario debe conceder los accesos para que determinados usuarios tengan permisos. Un proyecto *Interno* es visible a cualquier usuario identificado, y un proyecto *Público* es visible a todos, incluso usuarios identificados y visitantes. Observa que esto controla también el acceso de lectura git (“fetch”) así

como el acceso a la página web del proyecto.

Enganches (hooks)

GitLab tiene soporte para los enganches (hooks), tanto a nivel de proyecto como del sistema. Para cualquiera de ellos, el servidor GitLab realizará una petición HTTP POST con determinados datos JSON cuando ocurran determinados eventos. Es una manera interesante de conectar los repositorios y la instancia de GitLab con el resto de los mecanismos automáticos de desarrollo, como servidores de integración continua (CI), salas de charla y otras utilidades de despliegue.

Uso básico

Lo primero que tienes que hacer en GitLab es crear un nuevo proyecto. Esto lo consigues pulsando el icono "+" en la barra superior. Te preguntará por el nombre del proyecto, el espacio de nombres al que pertenece y qué nivel de visibilidad debe tener. Esta información, en su mayoría, no es fija y puedes cambiarla más tarde en la pantalla de ajustes. Pulsa en "Create Project" y habrás terminado.

Una vez que tengas el proyecto, querrás usarlo para un repositorio local de Git. Cada proyecto se puede acceder por HTTPS o SSH, protocolos que podemos configurar en nuestro repositorio como un Git remoto. La URL la encontrarás al principio de la página principal del proyecto. Para un repositorio local existente, puedes crear un remoto llamado `gitlab` del siguiente modo:

```
$ git remote add gitlab https://server/namespace/project.git
```

Si no tienes copia local del repositorio, puedes hacer esto:

```
$ git clone https://server/namespace/project.git
```

La interfaz web te permite acceder a diferentes vistas interesantes del repositorio. Además la página principal del proyecto muestra la actividad reciente, así como enlaces que permiten acceder a los archivos del proyecto y a los diferentes commits.

Trabajando con GitLab

Para trabajar en un proyecto GitLab lo más simple es tener acceso de escritura (push) sobre el repositorio git. Puedes añadir usuarios al proyecto en la sección "Members" de los ajustes del mismo, y asociar el usuario con un nivel de acceso (los niveles los hemos visto en Grupos). Cualquier nivel de acceso tipo "Developer" o superior permite al usuario enviar commits y ramas sin ninguna limitación.

Otra forma, más desacoplada, de colaboración, es mediante las peticiones de fusión (merge requests). Esta característica permite a cualquier usuario con acceso de lectura, participar de manera controlada. Los usuarios con acceso directo pueden, simplemente, crear la rama, enviar commits y luego abrir una petición de fusión desde su rama hacia la rama `master` u otra cualquiera. Los usuarios sin permiso de push pueden hacer un "fork" (es decir, su propia copia del repositorio), enviar sus cambios a *esa copia*, y abrir una petición de fusión desde su fork hacia el proyecto del que partió. Este modelo permite al propietario tener un control total de lo que entra en el

repositorio, permitiendo sin embargo la cooperación de usuarios en los que no se confía el acceso total.

Las peticiones de fusión y las incidencias (issues) son las principales fuentes de discusión en los proyectos de GitLab. Cada petición de fusión permite una discusión sobre el cambio propuesto (similar a una revisión de código), así como un hilo de discusión general. Ambas pueden asignarse a usuarios, o ser organizadas en hitos (milestones).

Esta sección se ha enfocado principalmente hacia las características de GitLab relacionadas con Git, pero como proyecto ya maduro, tiene muchas otras características para ayudar en la coordinación de grupos de trabajo, como wikis de proyecto y utilidades de mantenimiento. Una ventaja de GitLab es que, una vez que el servidor está configurado y funcionando, rara vez tendrás que tocar un archivo de configuración o acceder al servidor mediante SSH; casi toda la administración y uso se realizará mediante el navegador web.

Git en un alojamiento externo

Si no quieres realizar todo el trabajo que implica poner en marcha tu propio servidor Git, tienes varias opciones para alojar tus proyectos Git en un sitio externo dedicado. Esto tiene varias ventajas: normalmente en los alojamientos externos es fácil configurar y comenzar proyectos y no hay que preocuparse del mantenimiento del servidor o de su monitorización. Aunque pongas en marcha tu propio servidor internamente, probablemente quieras usar un sitio público para tu código abierto. Será más fácil que la comunidad de software libre encuentre tu proyecto y colabore.

Actualmente hay bastantes opciones de alojamiento para elegir, cada una con sus ventajas e inconvenientes. Para ver una lista actualizada, mira la página acerca de alojamiento Git en el wiki principal de Git, en https://git.wiki.kernel.org/index.php/GitHosting

Nos ocuparemos en detalle de Github en GitHub, al ser el sitio de alojamiento de proyectos más grande, y donde probablemente encuentres otros proyectos en los que quieras participar, pero en cualquier caso hay docenas de sitios para elegir sin necesidad de configurar tu propio servidor Git.

Resumen

Tienes varias opciones para obtener un repositorio Git remoto y ponerlo a funcionar para que puedas colaborar con otras personas o compartir tu trabajo.

Mantener tu propio servidor te da control y te permite correr tu servidor dentro de tu propio cortafuegos, pero tal servidor generalmente requiere una importante cantidad de tu tiempo para configurar y mantener. Si almacenas tus datos en un servidor hospedado, es fácil de configurar y mantener; sin embargo, tienes que ser capaz de mantener tu código en los servidores de alguien más, y algunas organizaciones no te lo permitirán.

Debería ser un proceso claro determinar que solución o combinación de soluciones es apropiada para ti y para tu organización.

Git en entornos distribuidos

Ahora que ya tienes un repositorio Git configurado como punto de trabajo para que los desarrolladores compartan su código, y además ya conoces los comandos básicos de Git para usar en local, verás cómo se puede utilizar alguno de los flujos de trabajo distribuido que Git permite.

En este capítulo verás como trabajar con Git en un entorno distribuido como colaborador o como integrador. Es decir, aprenderás como contribuir adecuadamente a un proyecto, de manera fácil tanto para ti como para el responsable del proyecto, y también como mantener adecuadamente un proyecto con múltiples desarrolladores.

Flujos de trabajo distribuidos

A diferencia de Sistemas Centralizados de Control de Versiones (CVCSs, Centralized Version Control Systems),la naturaleza distribuido de Git te permite mucha más flexibilidad en la manera de colaborar en proyectos. En los sistemas centralizados, cada desarrollador es un nodo de trabajo más o menos en igualdad con un repositorio central. En Git, sin embargo, cada desarrollador es potencialmente un nodo o un repositorio - es decir, cada desarrollador puede contribuir a otros repositorios y mantener un repositorio público en el cual otros pueden basar su trabajo y al cual pueden contribuir.

Esto abre un enorme rango de posibles flujos de trabajo para tu proyecto y/o tu equipo, así que revisaremos algunos de los paradigmas que toman ventajas de esta flexibilidad Repasaremos las fortalezas y posibles debilidades de cada diseño; podrás elegir uno solo o podrás mezclarlos para escoger características concretas de cada uno.

Flujos de trabajo centralizado

En sistemas centralizados, habitualmente solo hay un modelo de colaboración - el flujo de trabajo centralizado. Un repositorio o punto central que acepta código y todos sincronizan su trabajo con él. Unos cuantos desarrolladores son nodos de trabajo - consumidores de dicho repositorio - y sincronizan con ese punto.

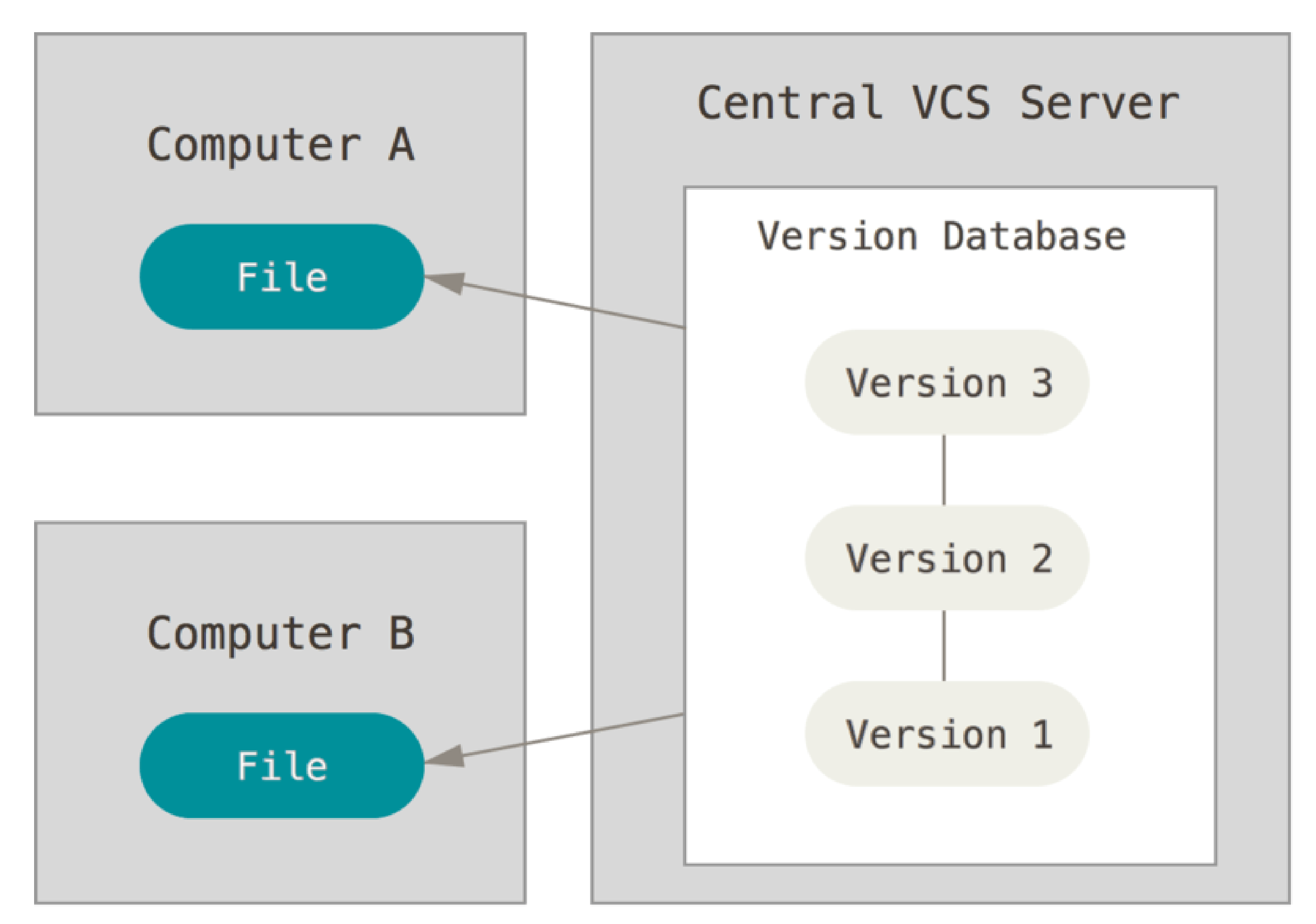


Figure 54. Centralized workflow.

Esto significa que si dos desarrolladores clonan desde el punto central, y ambos hacen cambios, solo el primer desarrollador en subir sus cambios lo podrá hacer sin problemas. El segundo desarrollador debe fusionar el trabajo del primero antes de subir sus cambios, para no sobreescribir los cambios del primero. Este concepto es válido tanto en Git como en Subversion This concept is as true in Git as it is in Subversion (or any CVCS), and this model works perfectly well in Git.

Si ya está cómodo con un flujo de trabajo centralizado en su empresa o en su equipo, puede seguir utilizando fácilmente ese flujo de trabajo con Git. Simplemente configure un único repositorio, y dé a cada uno en su equipo acceso de empuje; Git no permitirá que los usuarios se sobrescriban entre sí. Digamos que John y Jessica empiezan a trabajar al mismo tiempo. John termina su cambio y lo empuja al servidor. Entonces Jessica intenta empujar sus cambios, pero el servidor los rechaza. Le dicen que está tratando de empujar cambios no rápidos y que no podrá hacerlo hasta que busque y se fusione. Este flujo de trabajo es atractivo para mucha gente porque es un paradigma que muchos están familiarizados y cómodos.

Esto tampoco se limita a los equipos pequeños. Con el modelo de ramificación de Git, es posible que cientos de desarrolladores trabajen con éxito en un único proyecto a través de docenas de ramas simultáneamente.

Integration-Manager Workflow

Because Git allows you to have multiple remote repositories, it’s possible to have a workflow where each developer has write access to their own public repository and read access to everyone else’s.

This scenario often includes a canonical repository that represents the "official" project. To contribute to that project, you create your own public clone of the project and push your changes to it. Then, you can send a request to the maintainer of the main project to pull in your changes. The maintainer can then add your repository as a remote, test your changes locally, merge them into their branch, and push back to their repository. The process works as follows (see Integration-manager workflow.):

1. The project maintainer pushes to their public repository.
2. A contributor clones that repository and makes changes.
3. The contributor pushes to their own public copy.
4. The contributor sends the maintainer an e-mail asking them to pull changes.
5. The maintainer adds the contributor's repo as a remote and merges locally.
6. The maintainer pushes merged changes to the main repository.

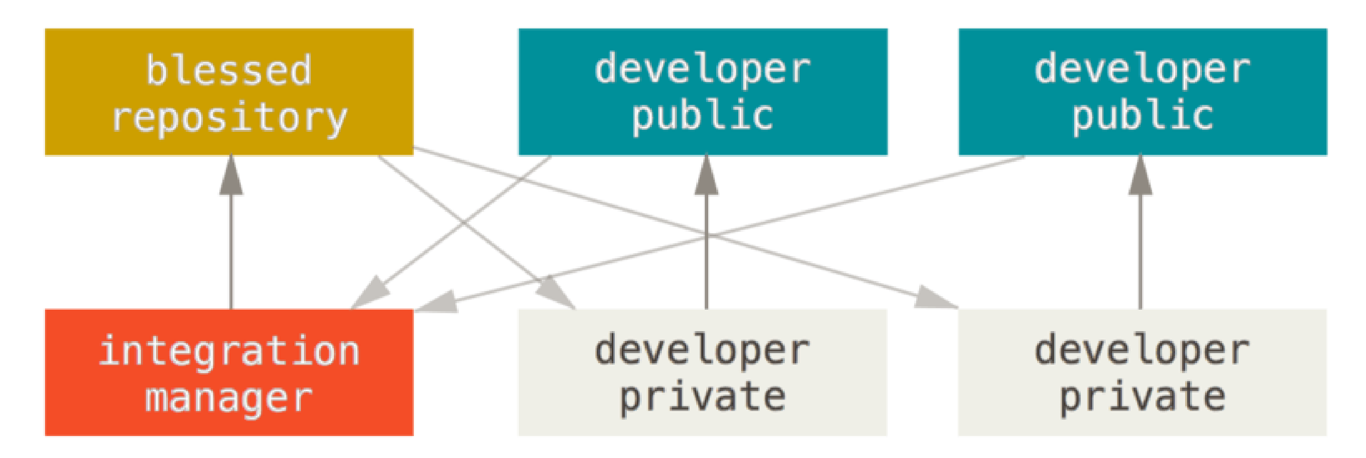


Figure 55. Integration-manager workflow.

This is a very common workflow with hub-based tools like GitHub or GitLab, where it's easy to fork a project and push your changes into your fork for everyone to see. One of the main advantages of this approach is that you can continue to work, and the maintainer of the main repository can pull in your changes at any time. Contributors don't have to wait for the project to incorporate their changes – each party can work at their own pace.

Dictator and Lieutenants Workflow

This is a variant of a multiple-repository workflow. It's generally used by huge projects with hundreds of collaborators; one famous example is the Linux kernel. Various integration managers are in charge of certain parts of the repository; they're called lieutenants. All the lieutenants have one integration manager known as the benevolent dictator. The benevolent dictator's repository serves as the reference repository from which all the collaborators need to pull. The process works like this (see Benevolent dictator workflow.):

1. Regular developers work on their topic branch and rebase their work on top of `master`. The `master` branch is that of the dictator.
2. Lieutenants merge the developers' topic branches into their `master` branch.
3. The dictator merges the lieutenants' `master` branches into the dictator's `master` branch.

4. The dictator pushes their `master` to the reference repository so the other developers can rebase on it.

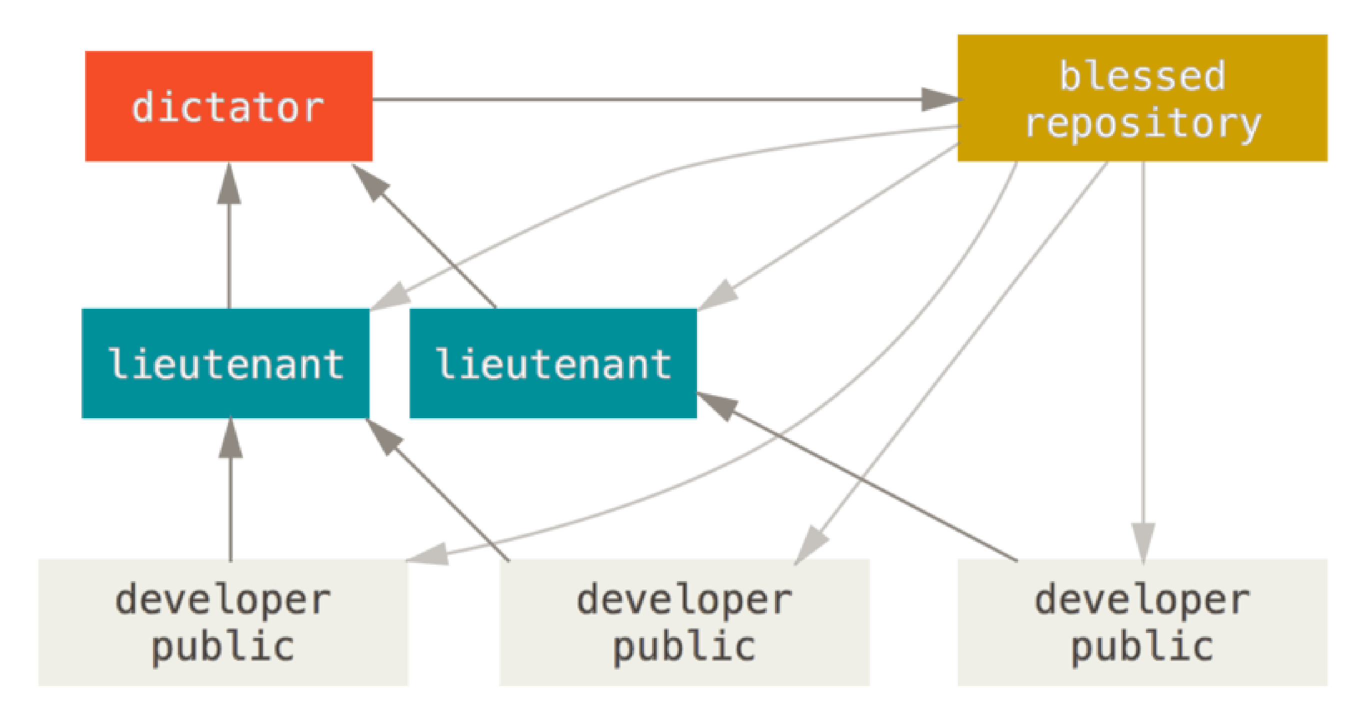


Figure 56. Benevolent dictator workflow.

This kind of workflow isn't common, but can be useful in very big projects, or in highly hierarchical environments. It allows the project leader (the dictator) to delegate much of the work and collect large subsets of code at multiple points before integrating them.

Workflows Summary

These are some commonly used workflows that are possible with a distributed system like Git, but you can see that many variations are possible to suit your particular real-world workflow. Now that you can (hopefully) determine which workflow combination may work for you, we'll cover some more specific examples of how to accomplish the main roles that make up the different flows. In the next section, you'll learn about a few common patterns for contributing to a project.

Contribuyendo a un Proyecto

La principal dificultad con la descripción de cómo contribuir a un proyecto es que hay una gran cantidad de variaciones sobre cómo se hace. Debido a que Git es muy flexible, las personas pueden y trabajan juntas de muchas maneras, y es problemático describir cómo debe contribuir: cada proyecto es un poco diferente. Algunas de las variables involucradas son conteo de contribuyentes activos, flujo de trabajo elegido, acceso de confirmación y posiblemente el método de contribución externa.

La primera variable es el conteo de contribuyentes activos: ¿cuántos usuarios están contribuyendo activamente al código de este proyecto y con qué frecuencia? En muchos casos, tendrá dos o tres desarrolladores con algunos commits por día, o posiblemente menos para proyectos un tanto inactivos. Para empresas o proyectos más grandes, la cantidad de desarrolladores podría ser de miles, con cientos o miles de compromisos cada día. Esto es importante porque con más y más desarrolladores, se encontrará con más problemas para asegurarse de que su código se aplique de

forma limpia o se pueda fusionar fácilmente. Los cambios que envíe pueden quedar obsoletos o severamente interrumpidos por el trabajo que se fusionó mientras estaba trabajando o mientras los cambios estaban esperando ser aprobados o aplicados. ¿Cómo puede mantener su código constantemente actualizado y sus confirmaciones válidas?

La siguiente variable es el flujo de trabajo en uso para el proyecto. ¿Está centralizado, con cada desarrollador teniendo el mismo acceso de escritura a la línea de código principal? ¿El proyecto tiene un mantenedor o un gerente de integración que verifica todos los parches? ¿Están todos los parches revisados por pares y aprobados? ¿Estás involucrado en ese proceso? ¿Hay un sistema de tenientes en su lugar, y tienes que enviar tu trabajo primero?

El siguiente problema es tu acceso de confirmación. El flujo de trabajo requerido para contribuir a un proyecto es muy diferente si tiene acceso de escritura al proyecto que si no lo hace. Si no tiene acceso de escritura, ¿cómo prefiere el proyecto aceptar el trabajo contribuido? ¿Incluso tiene una política? ¿Cuánto trabajo estás contribuyendo a la vez? ¿Con qué frecuencia contribuyes?

Todas estas preguntas pueden afectar la forma en que contribuye de manera efectiva a un proyecto y los flujos de trabajo preferidos o disponibles para usted. Cubriremos aspectos de cada uno de estos en una serie de casos de uso, pasando de simples a más complejos; debería poder construir los flujos de trabajo específicos que necesita en la práctica a partir de estos ejemplos.

Pautas de confirmación

Antes de comenzar a buscar casos de uso específicos, aquí hay una nota rápida sobre los mensajes de confirmación. Tener una buena guía para crear compromisos y apegarse a ella hace que trabajar con Git y colaborar con otros sea mucho más fácil. El proyecto Git proporciona un documento que presenta una serie de buenos consejos para crear compromisos a partir de los cuales enviar parches: puede leerlos en el código fuente de Git en el archivo `Documentation / SubmittingPatches`.

En primer lugar, no desea enviar ningún error de espacios en blanco. Git proporciona una manera fácil de verificar esto: antes de comprometerse, ejecute `git diff --check`, que identifica posibles errores de espacio en blanco y los enumera por usted.

```
bash
lib/simplegit.rb:5: trailing whitespace.
+    @git_dir = File.expand_path(git_dir)
lib/simplegit.rb:7: trailing whitespace.
+
lib/simplegit.rb:20: trailing whitespace.
+end
(END)
```

Figure 57. Output of `git diff --check`.

Si ejecuta ese comando antes de confirmar, puede ver si está a punto de cometer problemas de espacio en blanco que pueden molestar a otros desarrolladores.

A continuación, intente hacer que cada commit un conjunto de cambios lógicamente separado. Si puede, trate de hacer que sus cambios sean digeribles: no codifique durante un fin de semana entero en cinco asuntos diferentes y luego envíelos todos como un compromiso masivo el lunes. Incluso si no confirmas durante el fin de semana, utilice el área de etapas el lunes para dividir su trabajo en al menos una confirmación por cuestión, con un mensaje útil por confirmación. Si algunos de los cambios modifican el mismo archivo, intente utilizar `git add --patch` para representar parcialmente los archivos (se detalla en << _ interactive_staging >>). La instantánea del proyecto en la punta de la sucursal es idéntica ya sea que realice una confirmación o cinco, siempre que todos los cambios se agreguen en algún momento, así que trate de facilitar las cosas a sus compañeros desarrolladores cuando tengan que revisar sus cambios. Este enfoque también hace que sea más fácil retirar o revertir uno de los conjuntos de cambios si lo necesita más adelante. << _ rewriting_history >> describe una serie de útiles trucos de Git para reescribir el historial y organizar de forma interactiva los archivos: use estas herramientas para crear un historial limpio y comprensible antes de enviar el trabajo a otra persona.

Lo último a tener en cuenta es el mensaje de compromiso. Tener el hábito de crear mensajes de compromiso de calidad hace que usar y colaborar con Git sea mucho más fácil. Como regla general, sus mensajes deben comenzar con una sola línea que no supere los 50 caracteres y que describa el conjunto de cambios de forma concisa, seguido de una línea en blanco, seguida de una explicación más detallada. El proyecto Git requiere que la explicación más detallada incluya su motivación para el cambio y contraste su implementación con el comportamiento anterior: esta es una buena guía a seguir. También es una buena idea usar el tiempo presente imperativo en estos mensajes. En otras palabras, use comandos. En lugar de ‘` agregué pruebas para *’ o ‘` Añadir pruebas para ’*, use `` Agregar pruebas para ''. Aquí hay una plantilla escrita originalmente por Tim Pope:

```
Resumen de cambios cortos (50 caracteres o menos)

Texto explicativo más detallado, si es necesario. Ajustarlo a
aproximadamente 72 caracteres más o menos. En algunos contextos, el primero
línea se trata como el tema de un correo electrónico y el resto de
el texto como el cuerpo. La línea en blanco que separa el
resumen del cuerpo es crítico (a menos que omita el cuerpo
enteramente); herramientas como rebase pueden confundirse si ejecuta
los dos juntos.

Otros párrafos vienen después de las líneas en blanco.

  - Los puntos de viñetas también están bien

  - Típicamente se usa un guión o asterisco para la viñeta,
    precedido por un solo espacio, con líneas en blanco
    entre, pero las convenciones varían aquí
```

Si todos sus mensajes de confirmación se ven así, las cosas serán mucho más fáciles para usted y para los desarrolladores con los que trabaja. El proyecto Git tiene mensajes de confirmación bien formateados. Intenta ejecutar `git log --no-merges` allí para ver cómo se ve un historial de commit de proyecto muy bien formateado.

En los siguientes ejemplos, y en la mayor parte de este libro, en aras de la brevedad, este libro no tiene mensajes con un formato agradable como este; en cambio, usamos la opción `-m` para` git commit`. Haz lo que decimos, no como lo hacemos.

Pequeño equipo privado

La configuración más simple que es probable que encuentre es un proyecto privado con uno o dos desarrolladores más. ‘` Privado '', en este contexto, significa fuente cerrada, no accesible para el mundo exterior. Usted y los demás desarrolladores tienen acceso de inserción al repositorio.

En este entorno, puede seguir un flujo de trabajo similar a lo que podría hacer al usar Subversion u otro sistema centralizado. Aún obtiene las ventajas de cosas como el compromiso fuera de línea y una bifurcación y fusión mucho más simples, pero el flujo de trabajo puede ser muy similar; la principal diferencia es que las fusiones ocurren en el lado del cliente en lugar de en el servidor en el momento de la confirmación. Veamos cómo se vería cuando dos desarrolladores comiencen a trabajar junto con un repositorio compartido. El primer desarrollador, John, clona el repositorio, hace un cambio y se compromete localmente. (Los mensajes de protocolo se han reemplazado con `...` en estos ejemplos para acortarlos un poco).

```
# John's Machine
$ git clone john@githost:simplegit.git
Initialized empty Git repository in /home/john/simplegit/.git/
...
$ cd simplegit/
$ vim lib/simplegit.rb
$ git commit -am 'removed invalid default value'
[master 738ee87] removed invalid default value
 1 files changed, 1 insertions(+), 1 deletions(-)
```

El segundo desarrollador, Jessica, hace lo mismo: clona el repositorio y comete un cambio:

```
# Jessica's Machine
$ git clone jessica@githost:simplegit.git
Initialized empty Git repository in /home/jessica/simplegit/.git/
...
$ cd simplegit/
$ vim TODO
$ git commit -am 'add reset task'
[master fbff5bc] add reset task
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Ahora, Jessica lleva su trabajo al servidor:

```
# Jessica's Machine
$ git push origin master
...
To jessica@githost:simplegit.git
   1edee6b..fbff5bc  master -> master
```

John intenta impulsar su cambio, también:

```
# John's Machine
$ git push origin master
To john@githost:simplegit.git
 ! [rejected]        master -> master (non-fast forward)
error: failed to push some refs to 'john@githost:simplegit.git'
```

John no puede presionar porque Jessica ha presionado mientras tanto. Esto es especialmente importante para entender si está acostumbrado a Subversion, porque notará que los dos desarrolladores no editaron el mismo archivo. Aunque Subversion realiza automáticamente una fusión de este tipo en el servidor si se editan diferentes archivos, en Git debe fusionar los commit localmente. John tiene que buscar los cambios de Jessica y fusionarlos antes de que se le permita presionar:

```
$ git fetch origin
...
From john@githost:simplegit
 + 049d078...fbff5bc master     -> origin/master
```

En este punto, el repositorio local de John se ve así:

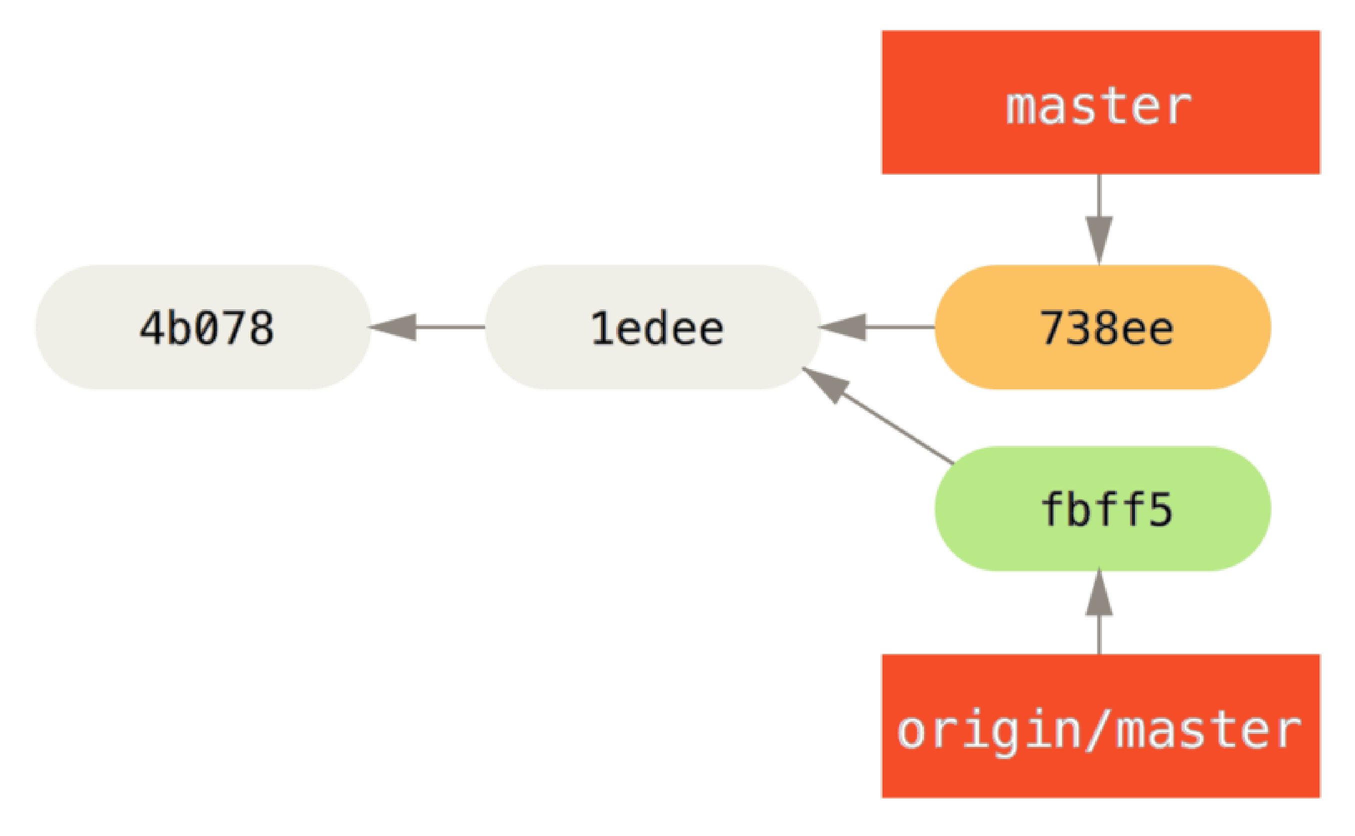


Figure 58. John's divergent history.

John tiene una referencia a los cambios que Jessica elevó, pero tiene que fusionarlos en su propio trabajo antes de que se le permita presionar:

```
$ git merge origin/master
Merge made by recursive.
 TODO |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

La fusión funciona sin problemas: el historial de compromisos de John ahora se ve así:

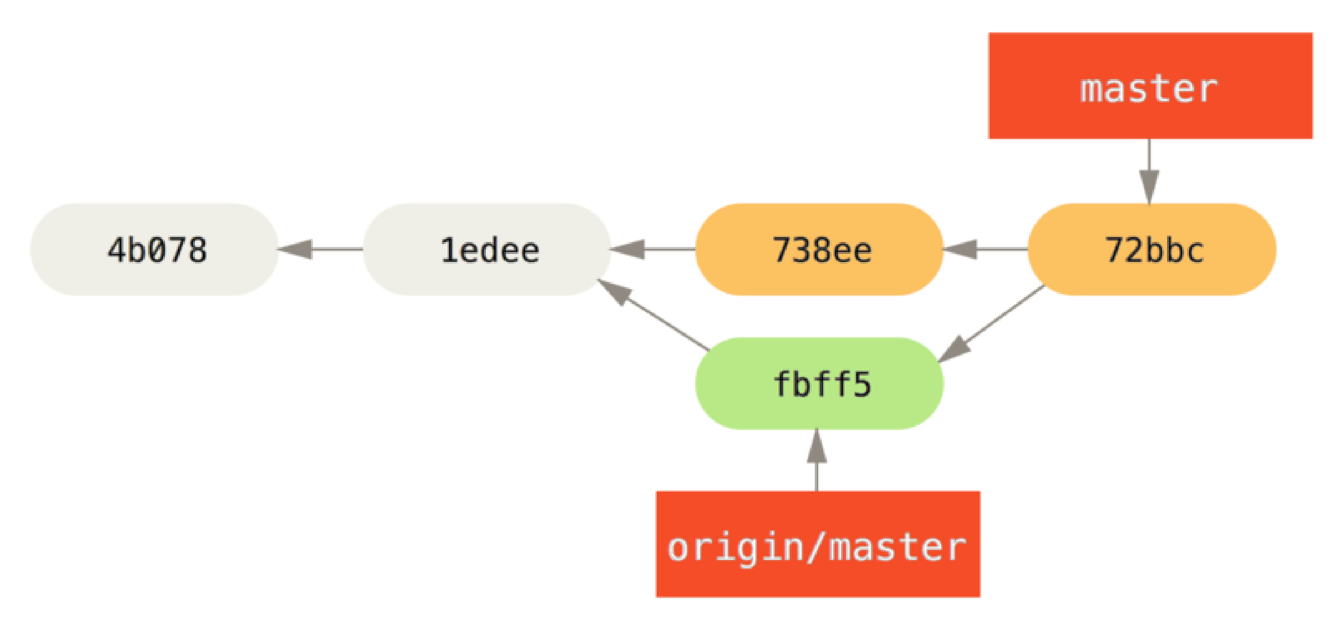


Figure 59. John's repository after merging `origin/master`.

Ahora, John puede probar su código para asegurarse de que todavía funciona correctamente, y luego puede enviar su nuevo trabajo combinado al servidor:

```
$ git push origin master
...
To john@githost:simplegit.git
   fbff5bc..72bbc59  master -> master
```

Finalmente, el historial de compromisos de John se ve así:

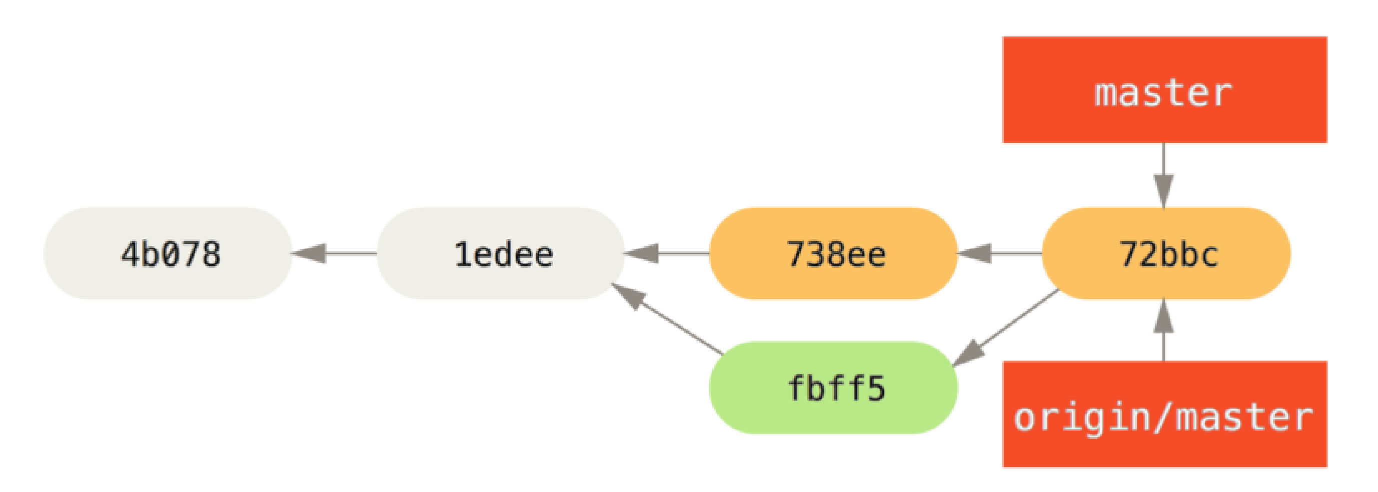


Figure 60. John's history after pushing to the `origin` server.

Mientras tanto, Jessica ha estado trabajando en una rama temática. Ella creó una rama temática llamada `issue54` y realizó tres commits en esa rama. Todavía no ha revisado los cambios de John, por lo que su historial de commit se ve así:

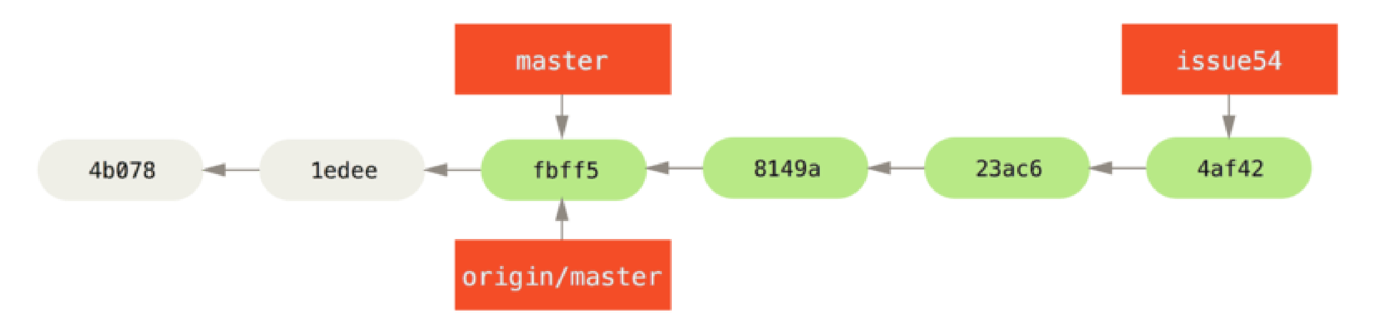


Figure 61. Jessica's topic branch.

Jessica quiere sincronizar con John, así que busca:

```
# Jessica's Machine
$ git fetch origin
...
From jessica@githost:simplegit
   fbff5bc..72bbc59  master     -> origin/master
```

Eso reduce el trabajo que John ha impulsado mientras tanto. La historia de Jessica ahora se ve así:

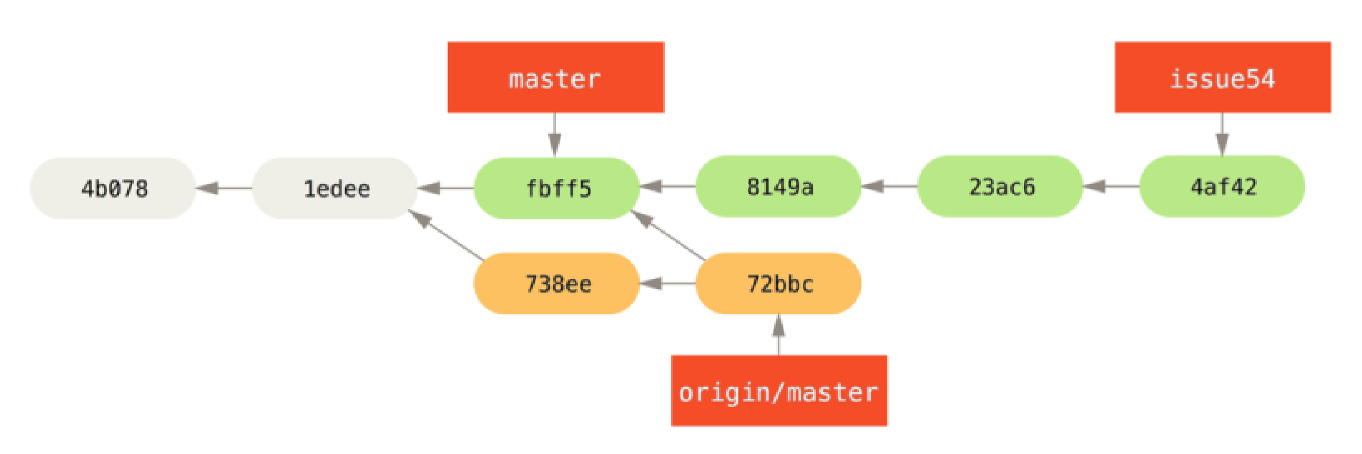


Figure 62. Jessica's history after fetching John's changes.

Jessica cree que su rama temática está lista, pero quiere saber qué tiene que fusionar en su trabajo para poder impulsarla. Ella ejecuta `git log` para descubrir:

```
$ git log --no-merges issue54..origin/master
commit 738ee872852dfaa9d6634e0dea7a324040193016
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 16:01:27 2009 -0700

   removed invalid default value
```

La sintaxis `issue54..origin / master` es un filtro de registro que le pide a Git que solo muestre la lista de confirmaciones que están en la última rama (en este caso` origen / maestro`) que no están en la primera rama (en este case `issue54`). Repasaremos esta sintaxis en detalle en << _ commit_ranges >>.

Por ahora, podemos ver en el resultado que hay un compromiso único que John ha realizado y en el

que Jessica no se ha fusionado. Si fusiona `origen / maestro`, esa es la única confirmación que modificará su trabajo local.

Ahora, Jessica puede fusionar su trabajo temático en su rama principal, fusionar el trabajo de John (`origin / master`) en su rama` master`, y luego volver al servidor nuevamente. Primero, vuelve a su rama principal para integrar todo este trabajo:

```
$ git checkout master
Switched to branch 'master'
Your branch is behind 'origin/master' by 2 commits, and can be fast-forwarded.
```

Ella puede fusionar ya sea `origin / master` o` issue54` primero - ambos están en sentido ascendente, por lo que el orden no importa. La instantánea final debe ser idéntica sin importar qué orden ella elija; solo la historia será ligeramente diferente. Ella elige fusionarse en `issue54` primero:

```
$ git merge issue54
Updating fbff5bc..4af4298
Fast forward
 README           |    1 +
 lib/simplegit.rb |    6 +++++-
 2 files changed, 6 insertions(+), 1 deletions(-)
```

No hay problemas como pueden ver, fue un simple avance rápido. Ahora Jessica se fusiona en el trabajo de John (`origin / master`):

```
$ git merge origin/master
Auto-merging lib/simplegit.rb
Merge made by recursive.
 lib/simplegit.rb |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Todo se funde limpiamente, y la historia de Jessica se ve así:

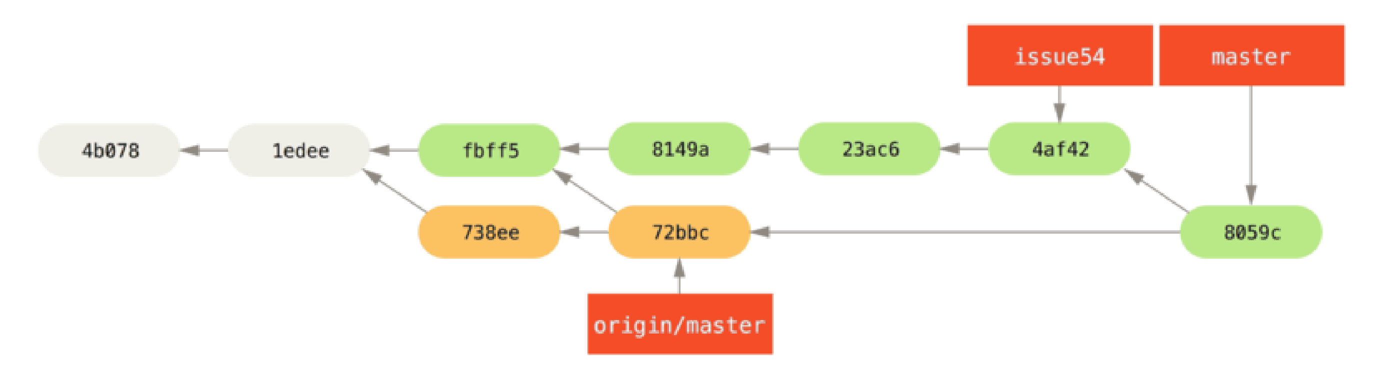


Figure 63. Jessica's history after merging John's changes.

Ahora `origin / master` es accesible desde la rama` master` de Jessica, por lo que debería poder presionar con éxito (suponiendo que John no haya pulsado nuevamente mientras tanto):

```
$ git push origin master
...
To jessica@githost:simplegit.git
   72bbc59..8059c15  master -> master
```

Cada desarrollador se ha comprometido algunas veces y se ha fusionado el trabajo de cada uno con éxito.

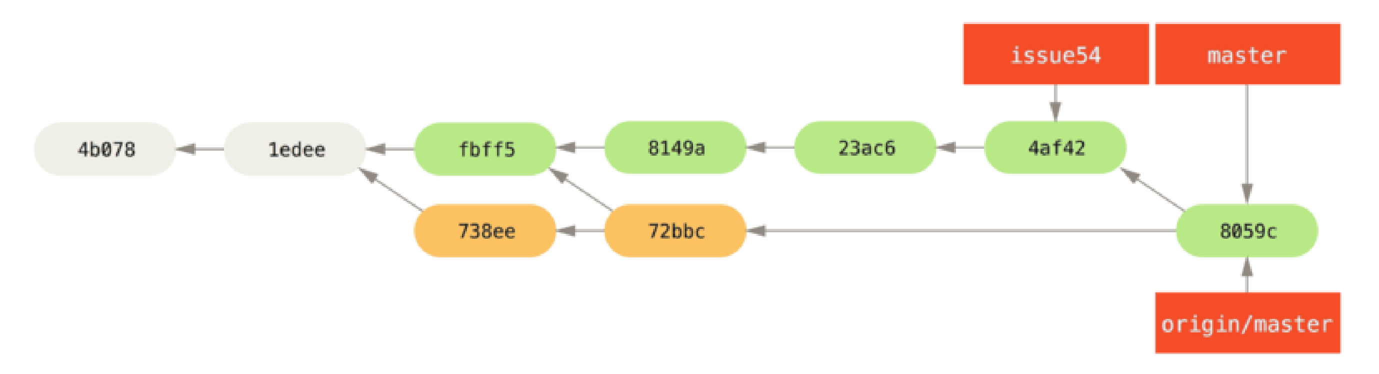


Figure 64. Jessica's history after pushing all changes back to the server.

Ese es uno de los flujos de trabajo más simples. Trabajas por un tiempo, generalmente en una rama temática, y te unes a tu rama principal cuando está lista para integrarse. Cuando desee compartir ese trabajo, hágalo en su propia rama principal, luego busque y combine `origin / master` si ha cambiado, y finalmente presione en la rama` master` en el servidor. La secuencia general es algo como esto:

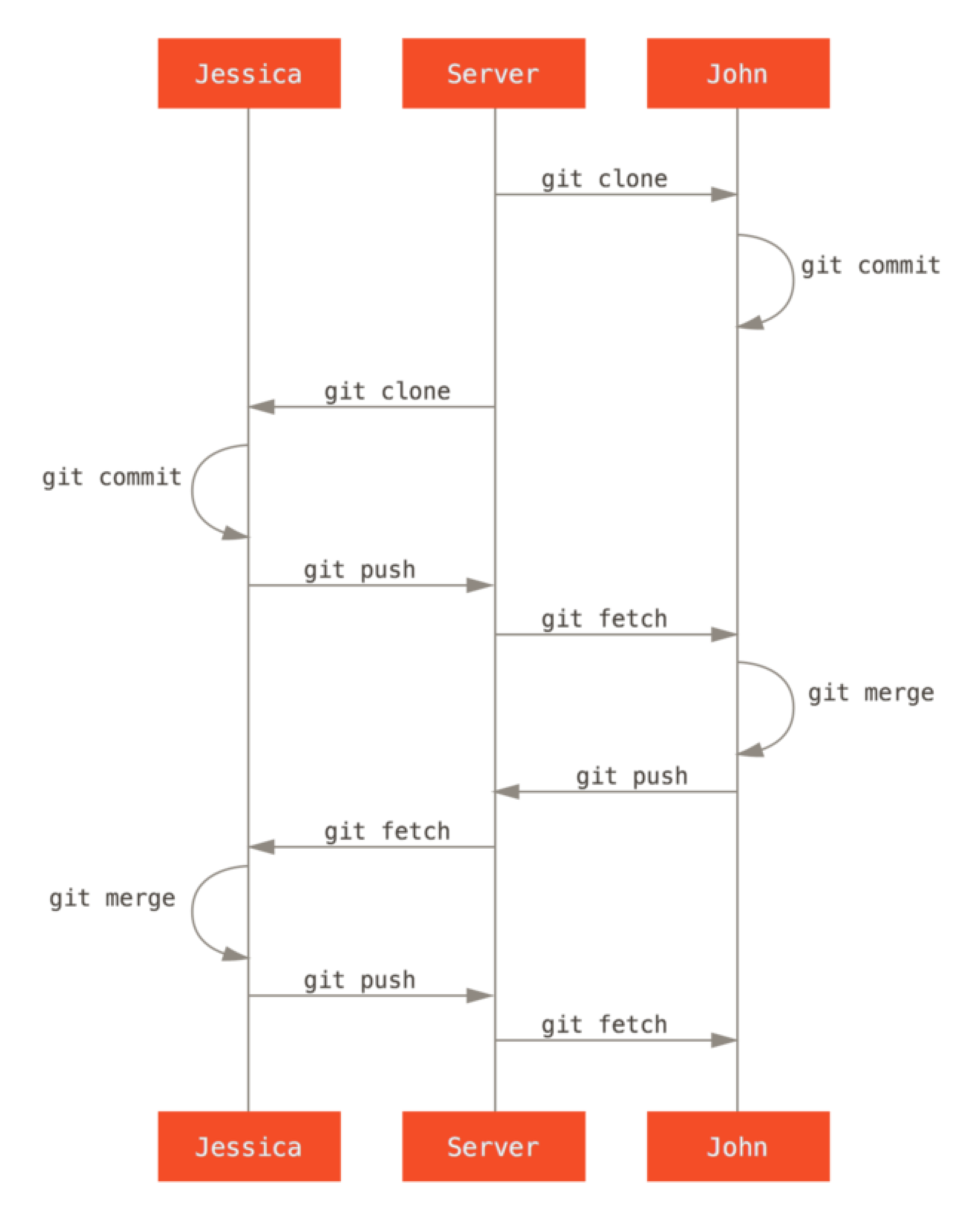


Figure 65. General sequence of events for a simple multiple-developer Git workflow.

Equipo privado administrado

En este próximo escenario, observará los roles de los contribuyentes en un grupo privado más grande. Aprenderá cómo trabajar en un entorno en el que los grupos pequeños colaboran en las funciones y luego esas contribuciones en equipo estarán integradas por otra parte.

Digamos que John y Jessica están trabajando juntos en una característica, mientras que Jessica y Josie están trabajando en una segunda. En este caso, la compañía está utilizando un tipo de flujo de

trabajo de integración-gerente donde el trabajo de los grupos individuales está integrado solo por ciertos ingenieros, y la rama `maestra` del repositorio principal solo puede ser actualizada por esos ingenieros. En este escenario, todo el trabajo se realiza en sucursales basadas en equipos y luego los integradores lo agrupan.

Sigamos el flujo de trabajo de Jessica mientras trabaja en sus dos funciones, colaborando en paralelo con dos desarrolladores diferentes en este entorno. Suponiendo que ya haya clonado su repositorio, ella decide trabajar primero en `featureA`. Ella crea una nueva sucursal para la característica y hace algo de trabajo allí:

```
# Jessica's Machine
$ git checkout -b featureA
Switched to a new branch 'featureA'
$ vim lib/simplegit.rb
$ git commit -am 'add limit to log function'
[featureA 3300904] add limit to log function
 1 files changed, 1 insertions(+), 1 deletions(-)
```

En este punto, ella necesita compartir su trabajo con John, por lo que empuja a su rama `featureA` a comprometerse con el servidor. Jessica no tiene acceso de inserción a la rama `maestra`, solo los integradores lo hacen, por lo que debe enviar a otra sucursal para colaborar con John:

```
$ git push -u origin featureA
...
To jessica@githost:simplegit.git
 * [new branch]      featureA -> featureA
```

Jessica le envía un correo electrónico a John para decirle que ha enviado algo de trabajo a una sucursal llamada `featureA` y ahora puede verlo. Mientras espera los comentarios de John, Jessica decide comenzar a trabajar en `featureB` con Josie. Para comenzar, inicia una nueva rama de características, basándose en la rama `master` del servidor:

```
# Jessica's Machine
$ git fetch origin
$ git checkout -b featureB origin/master
Switched to a new branch 'featureB'
```

Ahora, Jessica hace un par de commits en la rama `featureB`:

```
$ vim lib/simplegit.rb
$ git commit -am 'made the ls-tree function recursive'
[featureB e5b0fdc] made the ls-tree function recursive
 1 files changed, 1 insertions(+), 1 deletions(-)
$ vim lib/simplegit.rb
$ git commit -am 'add ls-files'
[featureB 8512791] add ls-files
 1 files changed, 5 insertions(+), 0 deletions(-)
```

El repositorio de Jessica se ve así:

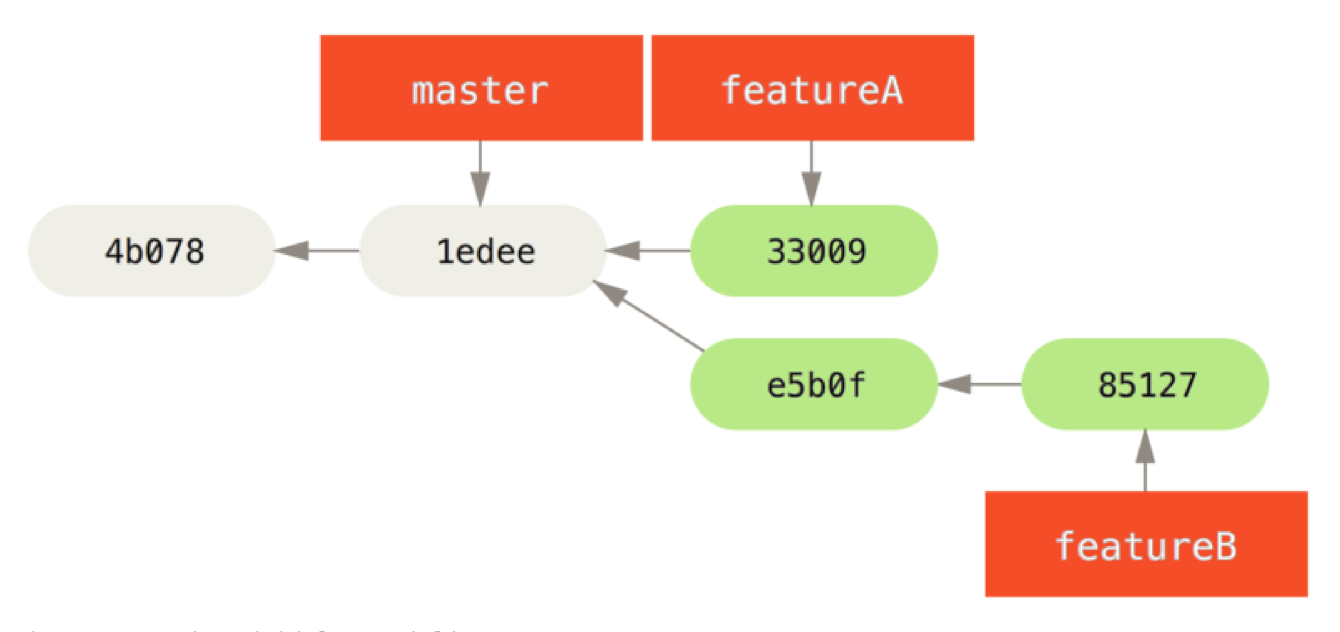


Figure 66. Jessica's initial commit history.

Está lista para impulsar su trabajo, pero recibe un correo electrónico de Josie que indica que una rama con algunos trabajos iniciales ya fue enviada al servidor como `featureBee`. Jessica primero necesita fusionar esos cambios con los suyos antes de poder presionar al servidor. Luego puede buscar los cambios de Josie con `git fetch`:

```
$ git fetch origin
...
From jessica@githost:simplegit
 * [new branch]      featureBee -> origin/featureBee
```

Jessica ahora puede fusionar esto en el trabajo que hizo con `git merge`:

```
$ git merge origin/featureBee
Auto-merging lib/simplegit.rb
Merge made by recursive.
 lib/simplegit.rb |    4 ++++
 1 files changed, 4 insertions(+), 0 deletions(-)
```

Existe un pequeño problema: necesita insertar el trabajo combinado en su rama `featureB` en la

rama` featureBee` del servidor. Ella puede hacerlo especificando la rama local seguida de dos puntos (:) seguido de la rama remota al comando `git push`:

```
$ git push -u origin featureB:featureBee
...
To jessica@githost:simplegit.git
   fba9af8..cd685d1  featureB -> featureBee
```

Esto se llama *refspec*. Ver << _ refspec >> para una discusión más detallada de las referencias de Git y diferentes cosas que puedes hacer con ellas. También observe la bandera `-u`; esto es la abreviatura de `--set-upstream`, que configura las ramas para empujar y tirar más fácilmente más tarde.

Luego, John envía un correo electrónico a Jessica para decirle que ha enviado algunos cambios a la rama `featureA` y le pide que los verifique. Ella ejecuta un `git fetch` para bajar esos cambios:

```
$ git fetch origin
...
From jessica@githost:simplegit
   3300904..aad881d  featureA   -> origin/featureA
```

Luego, ella puede ver qué se ha cambiado con `git log`:

```
$ git log featureA..origin/featureA
commit aad881d154acdaeb2b6b18ea0e827ed8a6d671e6
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 19:57:33 2009 -0700

    changed log output to 30 from 25
```

Finalmente, fusiona el trabajo de John en su propia rama `featureA`:

```
$ git checkout featureA
Switched to branch 'featureA'
$ git merge origin/featureA
Updating 3300904..aad881d
Fast forward
 lib/simplegit.rb |   10 +++++++++-
1 files changed, 9 insertions(+), 1 deletions(-)
```

Jessica quiere modificar algo, por lo que se compromete de nuevo y luego lo envía de vuelta al servidor:

```
$ git commit -am 'small tweak'
[featureA 774b3ed] small tweak
 1 files changed, 1 insertions(+), 1 deletions(-)
$ git push
...
To jessica@githost:simplegit.git
   3300904..774b3ed  featureA -> featureA
```

El historial de compromiso de Jessica ahora se ve así:

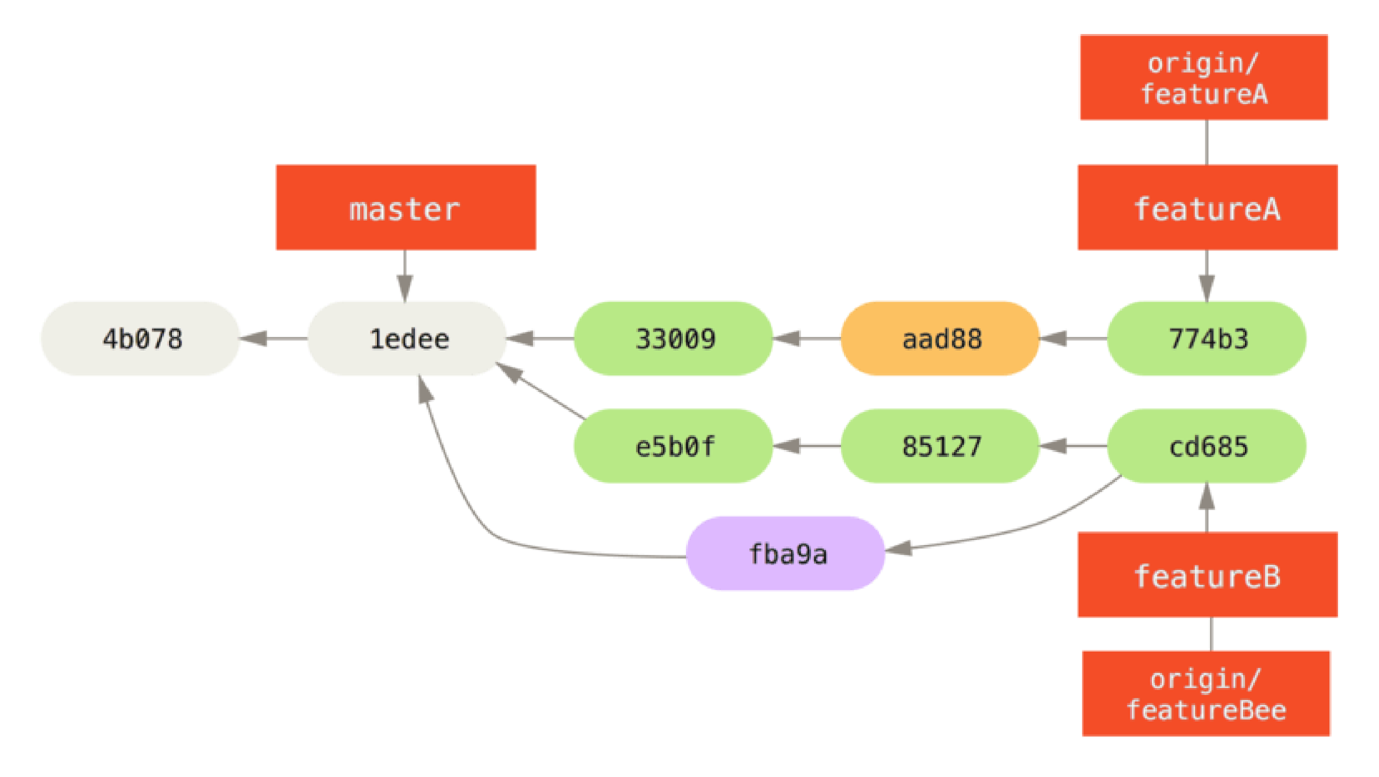


Figure 67. Jessica's history after committing on a feature branch.

Jessica, Josie y John informan a los integradores que las ramas `featureA` y` featureBee` en el servidor están listas para su integración en la línea principal. Después de que los integradores fusionen estas ramas en la línea principal, una búsqueda reducirá la nueva confirmación de fusión, haciendo que el historial se vea así:

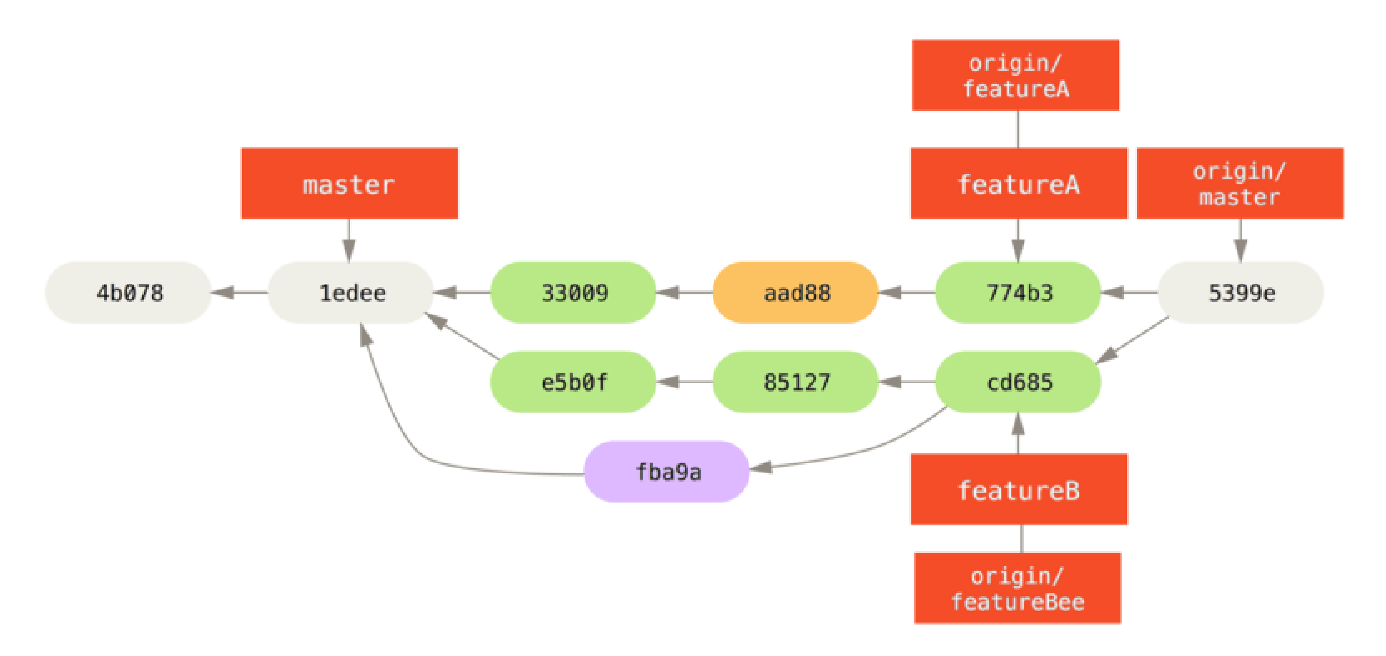


Figure 68. Jessica's history after merging both her topic branches.

Muchos grupos cambian a Git debido a esta capacidad de tener varios equipos trabajando en paralelo, fusionando las diferentes líneas de trabajo al final del proceso. La capacidad de los subgrupos más pequeños de un equipo para colaborar a través de sucursales remotas sin necesariamente tener que involucrar o impedir a todo el equipo es un gran beneficio de Git. La secuencia del flujo de trabajo que vio aquí es algo como esto:

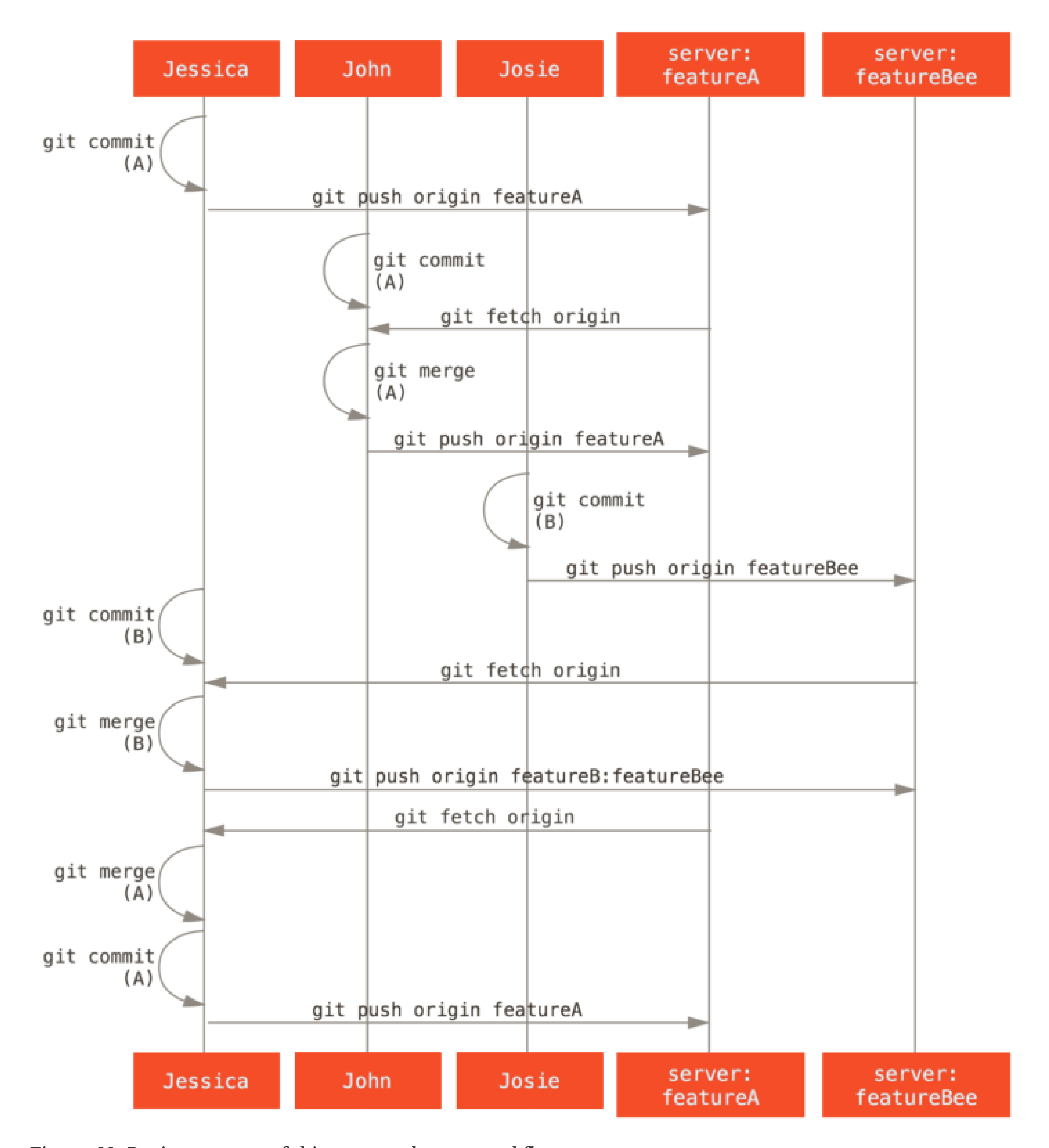


Figure 69. Basic sequence of this managed-team workflow.

Proyecto público bifurcado

Contribuir a proyectos públicos es un poco diferente. Como no tiene los permisos para actualizar directamente las ramas en el proyecto, debe obtener el trabajo de otra manera. Este primer ejemplo describe la contribución mediante bifurcación en hosts Git que admiten bifurcación fácil. Muchos sitios de alojamiento admiten esto (incluidos GitHub, BitBucket, Google Code, repo.or.cz y otros), y muchos mantenedores de proyectos esperan este estilo de contribución. La siguiente sección trata de proyectos que prefieren aceptar parches contribuidos por correo electrónico.

En primer lugar, es probable que desee clonar el repositorio principal, crear una rama de tema para el parche o la serie de parches que planea contribuir y hacer su trabajo allí. La secuencia se ve

básicamente así:

```
$ git clone (url)
$ cd project
$ git checkout -b featureA
# (work)
$ git commit
# (work)
$ git commit
```

NOTE Puede usar `rebase -i` para reducir su trabajo a una única confirmación, o reorganizar el trabajo en las confirmaciones para que el desarrollador pueda revisar el parche más fácilmente. Consulte << _ rewriting_history >> para obtener más información sobre el rebase interactivo.

Cuando finalice el trabajo de sucursal y esté listo para contribuir con los mantenedores, vaya a la página original del proyecto y haga clic en el botón ‘` Tenedor '', creando su propio tenedor escribible del proyecto. Luego debe agregar este nuevo URL de repositorio como segundo control remoto, en este caso llamado `myfork`:

```
$ git remote add myfork (url)
```

Entonces necesitas impulsar tu trabajo hasta eso. Es más fácil impulsar la rama de tema en la que está trabajando hasta su repositorio, en lugar de fusionarse con su rama principal y aumentarla. La razón es que si el trabajo no se acepta o se selecciona con una cereza, no es necesario rebobinar la rama maestra. Si los mantenedores se fusionan, redimensionan o seleccionan su trabajo, eventualmente lo recuperará a través de su repositorio de todas maneras:

```
$ git push -u myfork featureA
```

Cuando su trabajo ha sido empujado hacia su tenedor, debe notificar al mantenedor. Esto a menudo se denomina solicitud de extracción, y puedes generarlo a través del sitio web: GitHub tiene su propio mecanismo de solicitud de extracción que veremos en << _ github >> o puedes ejecutar el comando `git request-pull` y envíe por correo electrónico la salida al mantenedor del proyecto de forma manual.

El comando `request-pull` toma la rama base en la que desea que se saque su rama de tema y la URL del repositorio de Git de la que desea que extraigan, y genera un resumen de todos los cambios que está solicitando. Por ejemplo, si Jessica quiere enviar a John una solicitud de extracción, y ella ha hecho dos commits en la rama de temas que acaba de subir, puede ejecutar esto:

```
$ git request-pull origin/master myfork
The following changes since commit 1edee6b1d61823a2de3b09c160d7080b8d1b3a40:
  John Smith (1):
        added a new function

are available in the git repository at:

  git://githost/simplegit.git featureA

Jessica Smith (2):
      add limit to log function
      change log output to 30 from 25

 lib/simplegit.rb |   10 +++++++++-
 1 files changed, 9 insertions(+), 1 deletions(-)
```

La salida se puede enviar al mantenedor, les dice de dónde se ramificó el trabajo, resume los compromisos y dice de dónde sacar este trabajo.

En un proyecto para el cual no eres el mantenedor, generalmente es más fácil tener una rama como `master` siempre rastrea` origin / master` y hacer tu trabajo en ramas de tema que puedes descartar fácilmente si son rechazadas. Tener temas de trabajo aislados en las ramas temáticas también facilita la tarea de volver a establecer una base de trabajo si la punta del repositorio principal se ha movido mientras tanto y sus confirmaciones ya no se aplican limpiamente. Por ejemplo, si desea enviar un segundo tema de trabajo al proyecto, no continúe trabajando en la rama de tema que acaba de crear: vuelva a comenzar desde la rama `master` del repositorio principal:

```
$ git checkout -b featureB origin/master
# (work)
$ git commit
$ git push myfork featureB
# (email maintainer)
$ git fetch origin
```

Ahora, cada uno de sus temas está contenido dentro de un silo, similar a una fila de parches, que puede volver a escribir, volver a establecer y modificar sin que los temas interfieran o se interrelacionen entre sí, de la siguiente manera:

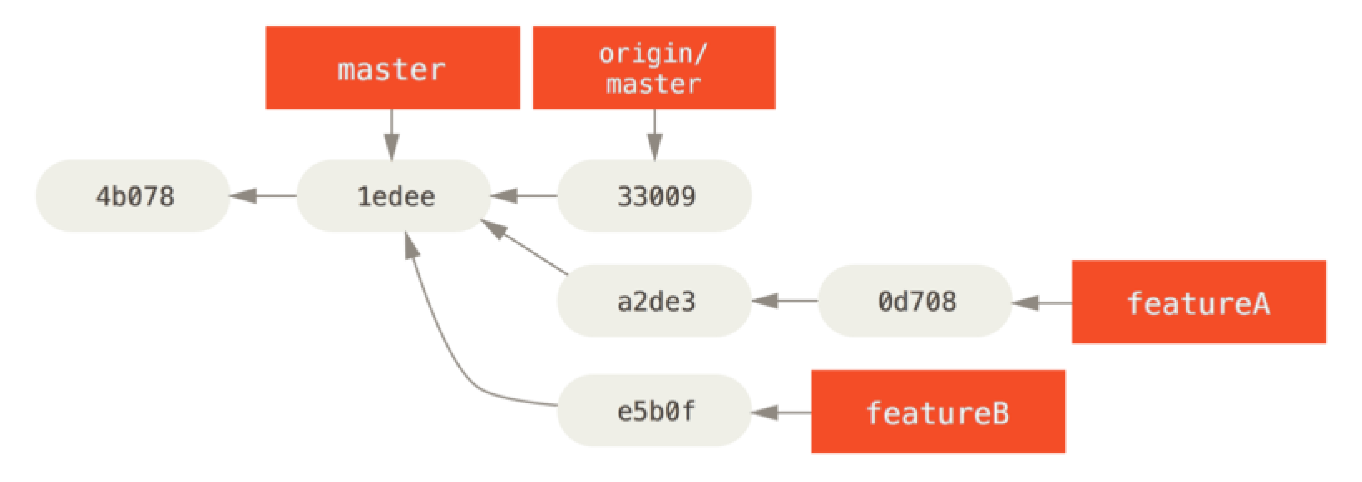


Figure 70. Initial commit history with `featureB` work.

Digamos que el mantenedor del proyecto ha sacado otros parches y ha probado su primera sucursal, pero ya no se fusiona de manera limpia. En este caso, puede tratar de volver a establecer la base de esa rama sobre *origin / master*, resolver los conflictos del mantenedor y luego volver a enviar los cambios:

```
$ git checkout featureA
$ git rebase origin/master
$ git push -f myfork featureA
```

Esto reescribe tu historial para que ahora parezca << psp_b >>.

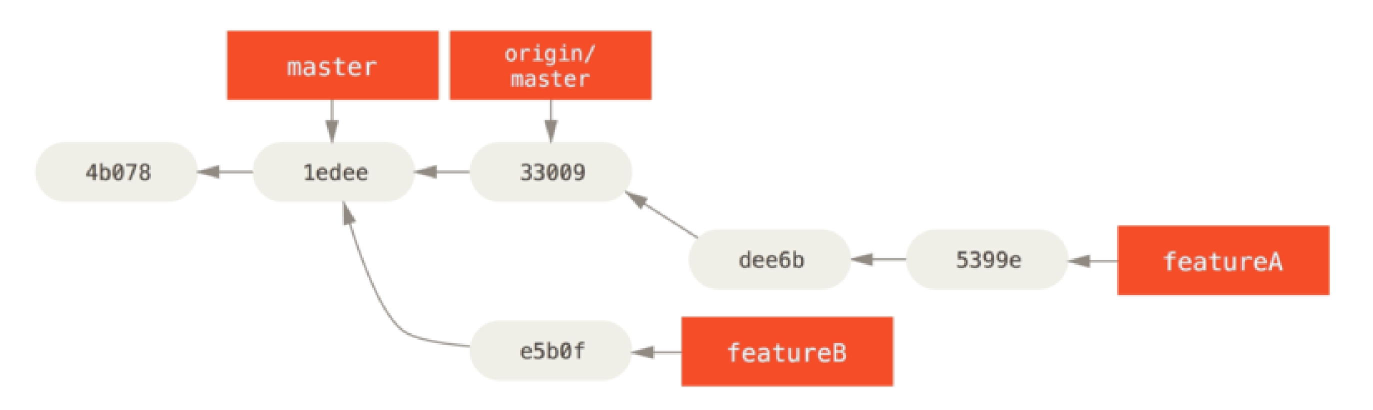


Figure 71. Commit history after `featureA` work.

Debido a que rebasaste la rama, debes especificar el `-f` en tu comando push para poder reemplazar la rama` featureA` en el servidor con una confirmación que no sea un descendiente de ella. Una alternativa sería llevar este nuevo trabajo a una rama diferente en el servidor (tal vez llamada `featureAv2`).

Veamos un escenario más posible: el mantenedor ha observado el trabajo en su segunda sucursal y le gusta el concepto, pero le gustaría que cambie un detalle de implementación. También aprovechará esta oportunidad para mover el trabajo basado en la rama `maestra 'actual del proyecto. Usted inicia una nueva sucursal basada en la rama actual de 'origen / maestro', aplasta los cambios `featureB` allí, resuelve cualquier conflicto, hace que la implementación cambie, y luego lo empuja hacia arriba como una nueva sucursal:

```
$ git checkout -b featureBv2 origin/master
$ git merge --no-commit --squash featureB
# (change implementation)
$ git commit
$ git push myfork featureBv2
```

La opción `--squash` toma todo el trabajo en la rama fusionada y lo aplasta en una confirmación sin fusión en la parte superior de la rama en la que se encuentra. La opción `--no-commit` le dice a Git que no registre automáticamente una confirmación. Esto le permite introducir todos los cambios desde otra rama y luego realizar más cambios antes de registrar la nueva confirmación.

Ahora puede enviar al mantenedor un mensaje de que ha realizado los cambios solicitados y puede encontrar esos cambios en su rama `featureBv2`.

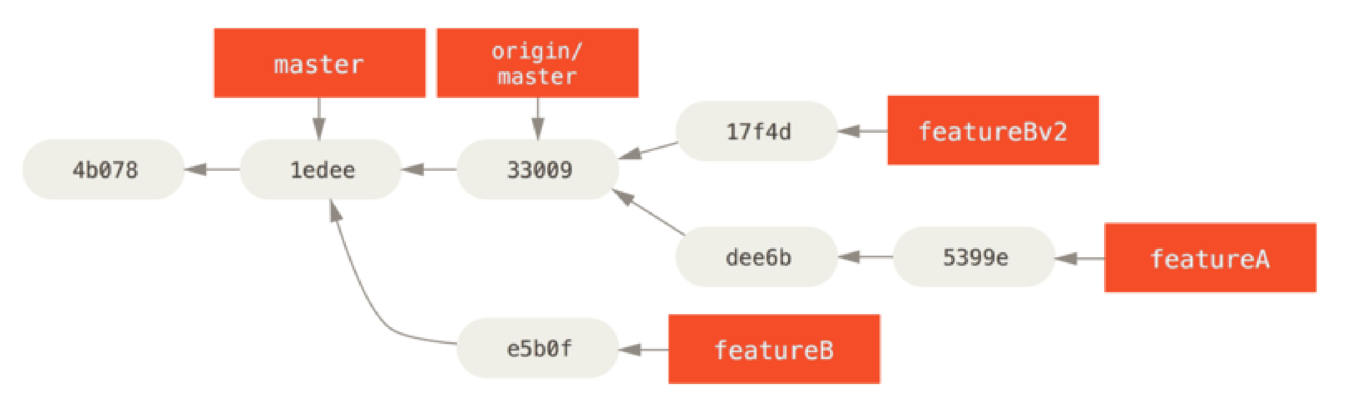


Figure 72. Commit history after `featureBv2` work.

Proyecto público a través de correo electrónico

Muchos proyectos han establecido procedimientos para aceptar parches: deberá verificar las reglas específicas para cada proyecto, ya que serán diferentes. Dado que hay varios proyectos antiguos y más grandes que aceptan parches a través de una lista de correo electrónico para desarrolladores, veremos un ejemplo de eso ahora.

El flujo de trabajo es similar al caso de uso anterior: crea ramas de tema para cada serie de parches en las que trabaja. La diferencia es cómo los envías al proyecto. En lugar de bifurcar el proyecto y avanzar hacia su propia versión de escritura, genera versiones de correo electrónico de cada serie de commit y las envía por correo electrónico a la lista de correo de desarrolladores:

```
$ git checkout -b topicA
# (work)
$ git commit
# (work)
$ git commit
```

Ahora tiene dos confirmaciones que desea enviar a la lista de correo. Utiliza `git format-patch` para generar los archivos con formato mbox que puedes enviar por correo electrónico a la lista:

convierte cada confirmación en un mensaje de correo electrónico con la primera línea del mensaje de confirmación como tema y el resto de el mensaje más el parche que introduce el compromiso como el cuerpo. Lo bueno de esto es que la aplicación de un parche de un correo electrónico generado con `format-patch` conserva toda la información de compromiso correctamente.

```
$ git format-patch -M origin/master
0001-add-limit-to-log-function.patch
0002-changed-log-output-to-30-from-25.patch
```

El comando `format-patch` imprime los nombres de los archivos de parche que crea. El modificador `-M` le dice a Git que busque cambios de nombre. Los archivos terminan pareciéndose a esto:

```
$ cat 0001-add-limit-to-log-function.patch
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] add limit to log function

Limit log functionality to the first 20

---
 lib/simplegit.rb |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 76f47bc..f9815f1 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -14,7 +14,7 @@ class SimpleGit
   end

   def log(treeish = 'master')
-    command("git log #{treeish}")
+    command("git log -n 20 #{treeish}")
   end

   def ls_tree(treeish = 'master')
--
2.1.0
```

También puede editar estos archivos de parche para agregar más información para la lista de correo electrónico que no desea mostrar en el mensaje de confirmación. Si agrega texto entre la línea `---` y el comienzo del parche (la línea `diff - git`), los desarrolladores pueden leerlo; pero aplicar el parche lo excluye.

Para enviarlo por correo electrónico a una lista de correo, puede pegar el archivo en su programa de correo electrónico o enviarlo a través de un programa de línea de comandos. Pegar el texto a menudo causa problemas de formateo, especialmente con clientes ‘` más inteligentes '' que no

conservan líneas nuevas y otros espacios en blanco de manera apropiada. Afortunadamente, Git proporciona una herramienta para ayudarlo a enviar parches con formato correcto a través de IMAP, que puede ser más fácil para usted. Demostraremos cómo enviar un parche a través de Gmail, que es el agente de correo electrónico que mejor conocemos; puede leer instrucciones detalladas para una cantidad de programas de correo al final del archivo `Documentation / SubmittingPatches` antes mencionado en el código fuente de Git.

Primero, necesitas configurar la sección imap en tu archivo `~ / .gitconfig`. Puede establecer cada valor por separado con una serie de comandos `git config`, o puede agregarlos manualmente, pero al final su archivo de configuración debería verse más o menos así:

```
[imap]
  folder = "[Gmail]/Drafts"
  host = imaps://imap.gmail.com
  user = user@gmail.com
  pass = p4ssw0rd
  port = 993
  sslverify = false
```

Si su servidor IMAP no usa SSL, las dos últimas líneas probablemente no sean necesarias, y el valor del host será `imap: //` en lugar de `imaps: //`. Cuando esté configurado, puede usar `git send-email` para colocar la serie de parches en la carpeta Borradores del servidor IMAP especificado:

```
$ git send-email *.patch
0001-added-limit-to-log-function.patch
0002-changed-log-output-to-30-from-25.patch
Who should the emails appear to be from? [Jessica Smith <jessica@example.com>]
Emails will be sent from: Jessica Smith <jessica@example.com>
Who should the emails be sent to? jessica@example.com
Message-ID to be used as In-Reply-To for the first email? y
```

Entonces, Git escupe un montón de información de registro que se ve algo así para cada parche que estás enviando:

```
(mbox) Adding cc: Jessica Smith <jessica@example.com> from
  \line 'From: Jessica Smith <jessica@example.com>'
OK. Log says:
Sendmail: /usr/sbin/sendmail -i jessica@example.com
From: Jessica Smith <jessica@example.com>
To: jessica@example.com
Subject: [PATCH 1/2] added limit to log function
Date: Sat, 30 May 2009 13:29:15 -0700
Message-Id: <1243715356-61726-1-git-send-email-jessica@example.com>
X-Mailer: git-send-email 1.6.2.rc1.20.g8c5b.dirty
In-Reply-To: <y>
References: <y>

Result: OK
```

En este punto, debe poder ir a la carpeta Borradores, cambiar el campo A a la lista de correo a la que le envía el parche, posiblemente CC al responsable o a la persona responsable de esa sección, y enviarlo.

Resumen

Esta sección ha cubierto una cantidad de flujos de trabajo comunes para tratar con varios tipos muy diferentes de proyectos de Git que probablemente encuentre, e introdujo algunas herramientas nuevas para ayudarlo a administrar este proceso. A continuación, verá cómo trabajar la otra cara de la moneda: mantener un proyecto de Git. Aprenderá cómo ser un dictador benevolente o un gerente de integración.

Manteniendo un proyecto

Además de saber cómo contribuir de manera efectiva a un proyecto, probablemente necesitarás saber cómo mantenerlo. Esto puede comprender desde aceptar y aplicar parches generados vía `format-patch` y enviados por e-mail, hasta integrar cambios en ramas remotas para repositorios que has añadido como remotos a tu proyecto. Tanto si mantienes un repositorio canónico como si quieres ayudar verificando o aprobando parches, necesitas conocer cómo aceptar trabajo de otros colaboradores de la forma lo más clara y sostenible posible a largo plazo.

Trabajando en ramas puntuales

Cuando estás pensando en integrar nuevo trabajo, generalmente es una buena idea probarlo en una rama puntual (topic branch) - una rama temporal específicamente creada para probar ese nuevo trabajo. De esta forma, es fácil ajustar un parche individualmente y abandonarlo si no funciona hasta que tengas tiempo de retomarlo. Si creas una rama simple con un nombre relacionado con el trabajo que vas a probar, como `ruby_client` o algo igualmente descriptivo, puedes recordarlo fácilmente si tienes que abandonarlo y retomarlo posteriormente. El responsable del mantenimiento del proyecto Git también tiende a usar una nomenclatura con estas ramas – como `sc/ruby_client`, donde `sc` es la abreviatura de la persona que envió el trabajo. Como recordarás, puedes crear la rama a partir de la rama master de la siguiente forma:

```
$ git branch sc/ruby_client master
```

O, si quieres cambiar inmediatamente a la rama, puedes usar la opción `checkout -b`:

```
$ git checkout -b sc/ruby_client master
```

Ahora estás listo para añadir el trabajo recibido en esta rama puntual y decidir si quieres incorporarlo en tus ramas de largo recorrido.

Aplicando parches recibidos por e-mail

Si recibes por e-mail un parche que necesitas integrar en tu proyecto, deberías aplicarlo en tu rama puntual para evaluarlo. Hay dos formas de aplicar un parche enviado por e-mail: con `git apply` o `git am`.

Aplicando un parche con apply

Si recibiste el parche de alguien que lo generó con `git diff` o con el comando Unix `diff` (lo cual no se recomienda; consulta la siguiente sección), puedes aplicarlo con el comando `git apply`. Suponiendo que guardaste el parche en `/tmp/patch-ruby-client.patch`, puedes aplicarlo de esta forma:

```
$ git apply /tmp/patch-ruby-client.patch
```

Esto modifica los archivos en tu directorio de trabajo. Es casi idéntico a ejecutar un comando `patch -p1` para aplicar el parche, aunque es más paranoico y acepta menos coincidencias aproximadas que patch. También puede manejar archivos nuevos, borrados y renombrados si están descritos en formato `git diff`, mientras que `patch` no puede hacerlo. Por último, `git apply` sigue un modelo "aplica todo o aborta todo", donde se aplica todo o nada, mientras que `patch` puede aplicar parches parcialemente, dejando tu directorio de trabajo en un estado inconsistente. `git apply` es en general mucho más conservador que `patch`. No creará un commit por ti – tras ejecutarlo, debes preparar (stage) y confirmar (commit) manualmente los cambios introducidos.

También puedes usar `git apply` para comprobar si un parche se aplica de forma limpia antes de aplicarlo realmente – puedes ejecutar `git apply --check` indicando el parche:

```
$ git apply --check 0001-seeing-if-this-helps-the-gem.patch
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
```

Si no obtienes salida, entonces el parche debería aplicarse limpiamente. Este comando también devuelve un estado distinto de cero si la comprobación falla, por lo que puedes usarlo en scripts.

Aplicando un parche con `am`

Si el colaborador es usuario de Git y conoce lo suficiente como para usar el comando `format-patch` para generar el parche, entonces tu trabajo es más sencillo, ya que el parche ya contiene información del autor y un mensaje de commit. Si puedes, anima a tus colaboradores a usar `format-patch` en lugar de `diff` para generar parches. Sólo deberías usar `git apply` para parches antiguos y cosas similares.

Para aplicar un parche generado con `format-patch`, usa `git am`. Técnicamente, `git am` se construyó para leer de un archivo mbox (buzón de correo). Es un formato de texto plano simple para almacenar uno o más mensajes de correo en un archivo de texto. Es algo parecido a esto:

```
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] add limit to log function

Limit log functionality to the first 20
```

Esto es el comienzo de la salida del comando `format-patch` que viste en la sección anterior. También es un formato mbox válido. Si alguien te ha enviado el parche por e-mail usando `git send-email` y lo has descargado en formato mbox, entonces puedes pasar ese archivo a `git am` y comenzará a aplicar todos los parches que encuentre. Si usas un cliente de correo que puede guardar varios e-mails en formato mbox, podrías guardar conjuntos completos de parches en un único archivo y a continuación usar `git am` para aplicarlos de uno en uno.

Sin embargo, si alguien subió a un sistema de gestión de incidencias o algo parecido un parche generado con `format-patch`, podrías guardar localmente el archivo y posteriormente pasarlo a `git am` para aplicarlo:

```
$ git am 0001-limit-log-function.patch
Applying: add limit to log function
```

Puedes ver que aplicó el parche limpiamente y creó automáticamente un nuevo commit. La información del autor se toma de las cabeceras `From` y `Date` del e-mail, y el mensaje del commit sale del `Subject` y el cuerpo del e-mail (antes del parche). Por ejemplo, si se aplicó este parche desde el archivo mbox del ejemplo anterior, el commit generado sería algo como esto:

```
$ git log --pretty=fuller -1
commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
Author:     Jessica Smith <jessica@example.com>
AuthorDate: Sun Apr 6 10:17:23 2008 -0700
Commit:     Scott Chacon <schacon@gmail.com>
CommitDate: Thu Apr 9 09:19:06 2009 -0700

   add limit to log function

   Limit log functionality to the first 20
```

El campo `Commit` indica la persona que aplicó el parche y cuándo lo aplicó. El campo `Author` es la persona que creó originalmente el parche y cuándo fue creado.

Pero es posible que el parche no se aplique limpiamente. Quizás tu rama principal es muy diferente de la rama a partir de la cual se creó el parche, o el parche depende de otro parche que aún no has aplicado. En ese caso, el proceso `git am` fallará y te preguntará qué hacer:

```
$ git am 0001-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Patch failed at 0001.
When you have resolved this problem run "git am --resolved".
If you would prefer to skip this patch, instead run "git am --skip".
To restore the original branch and stop patching run "git am --abort".
```

Este comando marca los conflictos en cualquier archivo para el cual detecte problemas, como si fuera una operación `merge` o `rebase`. Estos problemas se solucionan de la misma forma - edita el archivo para resolver el conflicto, prepara (stage) el nuevo archivo, y por último ejecuta `git am --resolved` para continuar con el siguiente parche:

```
$ (fix the file)
$ git add ticgit.gemspec
$ git am --resolved
Applying: seeing if this helps the gem
```

Si quieres que Git intente resolver el conflicto de forma un poco más inteligente, puedes indicar la opción `-3` para que Git intente hacer un merge a tres bandas. Esta opción no está activa por defecto, ya que no funciona si el commit en que el parche está basado no está en tu repositorio. Si tienes ese commit – si el parche partió de un commit público – entonces la opción `-3` es normalmente mucho más inteligente a la hora de aplicar un parche conflictivo:

```
$ git am -3 0001-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
No changes -- Patch already applied.
```

En este caso, el parche ya ha sido aplicado. Sin la opción `-3`, parecería un conflicto.

Si estás aplicando varios parches a partir de un archivo mbox, también puedes ejecutar el comando `am` en modo interactivo, el cual se detiene en cada parche para preguntar si quieres aplicarlo:

```
$ git am -3 -i mbox
Commit Body is:
--------------------------
seeing if this helps the gem
--------------------------
Apply? [y]es/[n]o/[e]dit/[v]iew patch/[a]ccept all
```

Esto está bien si tienes guardados varios parches, ya que puedes revisar el parche previamente y no aplicarlo si ya lo has hecho.

Una vez aplicados y confirmados todos los parches de tu rama puntual, puedes decidir cómo y cuándo integrarlos en una rama de largo recorrido.

Recuperando ramas remotas

Si recibes una contribución de un usuario de Git que configuró su propio repositorio, realizó cambios en él y envió la URL del repositorio junto con el nombre de la rama remota donde están los cambios, puedes añadirlo como una rama remota y hacer integraciones (merges) de forma local.

Por ejemplo, si Jessica te envía un e-mail diciendo que tiene una nueva funcionalidad muy interesante en la rama `ruby-client` de su repositorio, puedes probarla añadiendo el repositorio remoto y recuperando localmente dicha rama:

```
$ git remote add jessica git://github.com/jessica/myproject.git
$ git fetch jessica
$ git checkout -b rubyclient jessica/ruby-client
```

Si más tarde te envía otro email con una nueva funcionalidad en otra rama, puedes recuperarla (fetch y check out) directamente porque ya tienes el repositorio remoto configurado.

Esto es más útil cuando trabajas regularmente con una persona. Sin embargo, si alguien sólo envía un parche de forma ocasional, aceptarlo vía email podría llevar menos tiempo que obligar a todo el mundo a ejecutar su propio servidor y tener que añadir y eliminar repositorios remotos continuamente para obtener unos cuantos parches. Además, probablemente no quieras tener

cientos de repositorios remotos, uno por cada persona que envía uno o dos parches. En cualquier caso, los scripts y los servicios alojados pueden facilitar todo esto — depende en gran medida de cómo desarrollan tanto tus colaboradores como tú mismo —

Otra ventaja de esta opción es que además puedes obtener un historial de commits. Aunque pueden surgir los problemas habituales durante la integración (merge), al menos sabes en qué punto de tu historial se basa su trabajo; Por defecto, se realiza una integración a tres bandas, en lugar de indicar un `-3` y esperar que el parche se generara a partir de un commit público al que tengas acceso.

Si no trabajas regularmente con alguien pero aún así quieres obtener sus contribuciones de esta manera, puedes pasar la URL del repositorio remoto al comando `git pull`. Esto recupera los cambios de forma puntual (pull) sin guardar la URL como una referencia remota:

```
$ git pull https://github.com/onetimeguy/project
From https://github.com/onetimeguy/project
 * branch            HEAD       -> FETCH_HEAD
Merge made by recursive.
```

Decidiendo qué introducir

Ahora tienes una rama puntual con trabajo de los colaboradores. En este punto, puedes decidir qué quieres hacer con ella. Esta sección repasa un par de comandos para que puedas ver cómo se usan para revisar exactamente qué se va a introducir si integras los cambios en tu rama principal.

A menudo es muy útil obtener una lista de todos los commits de una rama que no están en tu rama principal. Puedes excluir de dicha lista los commits de tu rama principal anteponiendo la opción `--not` al nombre de la rama. El efecto de esto es el mismo que el formato `master..contrib` que usamos anteriormente. Por ejemplo, si un colaborador te envía dos parches y creas una rama llamada `contrib` para aplicar los parches, puedes ejecutar esto:

```
$ git log contrib --not master
commit 5b6235bd297351589efc4d73316f0a68d484f118
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Oct 24 09:53:59 2008 -0700

    seeing if this helps the gem

commit 7482e0d16d04bea79d0dba8988cc78df655f16a0
Author: Scott Chacon <schacon@gmail.com>
Date:   Mon Oct 22 19:38:36 2008 -0700

    updated the gemspec to hopefully work better
```

Para ver qué cambios introduce cada commit, recuerda que puedes indicar la opción `-p` a `git log`, y añadirá las diferencias introducidas en cada commit.

Para tener una visión completa de qué ocurriría si integraras esta rama puntual en otra rama,

podrías usar un sencillo truco para obtener los resultados correctos. Podrías pensar en ejecutar esto:

```
$ git diff master
```

Este comando te da las diferencias, pero los resultados podrían ser confusos. Si tu rama `master` ha avanzado desde que creaste la rama puntual, entonces obtendrás resultados aparentemente extraños. Esto ocurre porque Git compara directamente las instantáneas del último commit de la rama puntual en la que estás con la instantánea del último commit de la rama `master`. Por ejemplo, si has añadido una línea a un archivo en la rama `master`, al hacer una comparación directa de las instantáneas parecerá que la rama puntual va a eliminar esa línea.

Si `master` es un ancestro de tu rama puntual, esto no supone un problema; pero si los dos historiales divergen, al hacer una comparación directa parecerá que estás añadiendo todos los cambios nuevos en tu rama puntual y eliminándolos de la rama `master`.

Lo que realmente necesitas ver son los cambios añadidos en tu rama puntual – el trabajo que introducirás si integras esta rama en la `master`. Para conseguir esto, Git compara el último commit de tu rama puntual con el primer ancestro en común respecto a la rama `master`.

Técnicamente puedes hacer esto averiguando explícitamente el ancestro común y ejecutando el `diff` sobre dicho ancestro:

```
$ git merge-base contrib master
36c7dba2c95e6bbb78dfa822519ecfec6e1ca649
$ git diff 36c7db
```

Sin embargo, eso no es lo más conveniente, así que Git ofrece un atajo para hacer eso mismo: la sintaxis del triple-punto. En el contexto del comando `diff`, puedes poner tres puntos tras el nombre de una rama para hacer un `diff` entre el último commit de la rama en la que estás y su ancestro común con otra rama:

```
$ git diff master...contrib
```

Este comando sólo muestra el trabajo introducido en tu rama puntual actual desde su ancestro común con la rama `master`. Es una sintaxis muy útil a recordar.

Integrando el trabajo de los colaboradores

Cuando todo el trabajo de tu rama puntual está listo para ser integrado en una rama de largo recorrido, la cuestión es cómo hacerlo. Es más, ¿qué flujo de trabajo general quieres seguir para mantener el proyecto? Tienes varias opciones y vamos a ver algunas de ellas.

Integrando flujos de trabajo

Un flujo de trabajo sencillo integra tu trabajo en tu rama `master`. En este escenario, tienes una rama

master que contiene básicamente código estable. Cuando tienes trabajo propio en una rama puntual o trabajo aportado por algún colaborador que ya has verificado, lo integras en tu rama master, borras la rama puntual y continúas el proceso. Si tenemos un repositorio con trabajo en dos ramas llamadas ruby_client y php_client, tal y como se muestra en Historial con varias ramas puntuales. e integramos primero ruby_client y luego php_client, entonces tu historial terminará con este aspecto Tras integrar una rama puntual..

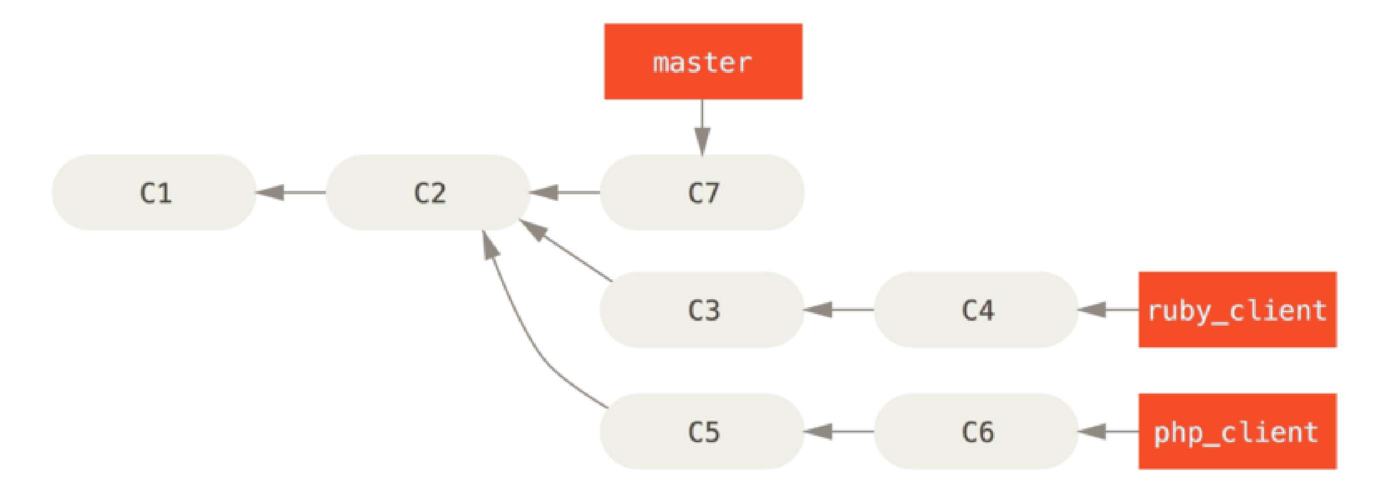


Figure 73. Historial con varias ramas puntuales.

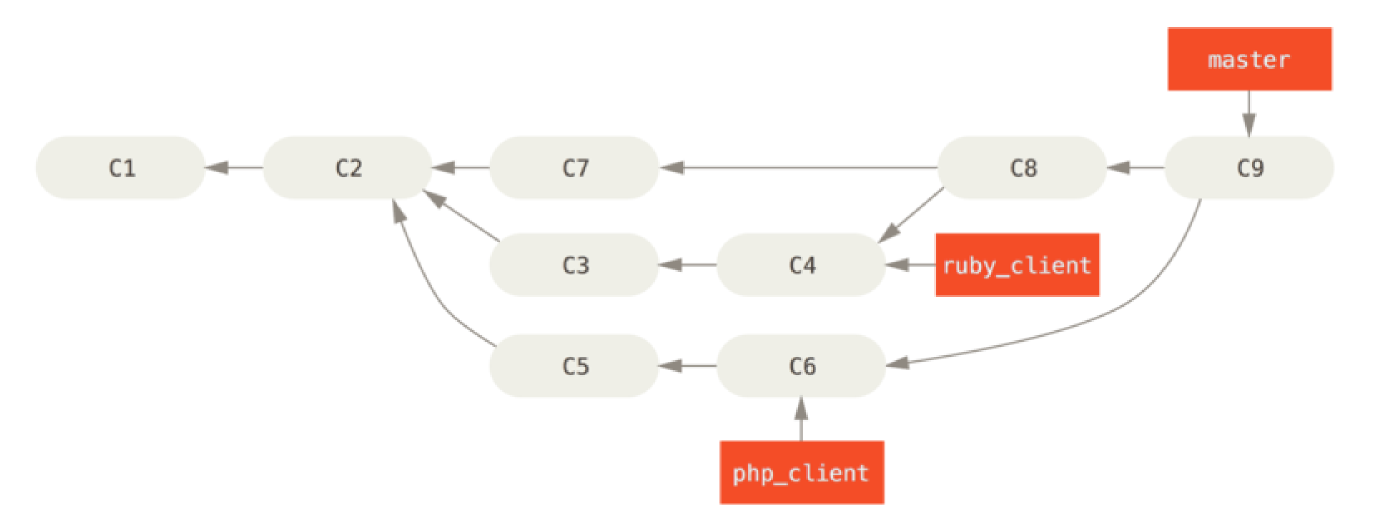


Figure 74. Tras integrar una rama puntual.

Este es probablemente el flujo de trabajo más simple y puede llegar a causar problemas si estás tratando con proyectos de mayor envergadura o más estables, donde hay que ser realmente cuidadoso al introducir cambios.

Si tienes un proyecto más importante, podrías preferir usar un ciclo de integración en dos fases. En este escenario, tienes dos ramas de largo recorrido, master y develop, y decides que la rama master sólo se actualiza cuando se llega a una versión muy estable y todo el código nuevo está integrado en la rama develop. Ambas ramas de envían habitualmente al repositorio público. Cada vez que tengas una nueva rama puntual para integrar en (Antes de integrar una rama puntual.), primero la fusionas con la rama develop (Tras integrar una rama puntual.); luego, tras etiquetar la versión, avanzas la rama master hasta el punto donde se encuentre la ahora estable rama develop (Tras el lanzamiento de una rama puntual.).

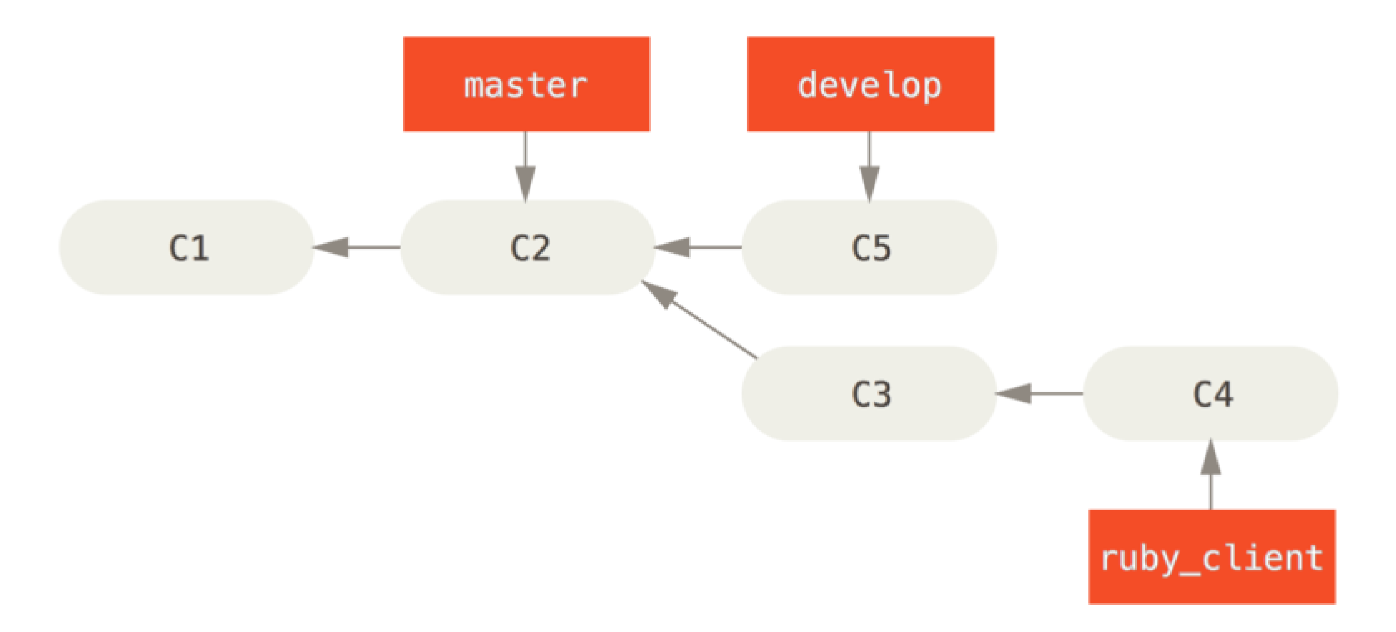


Figure 75. Antes de integrar una rama puntual.

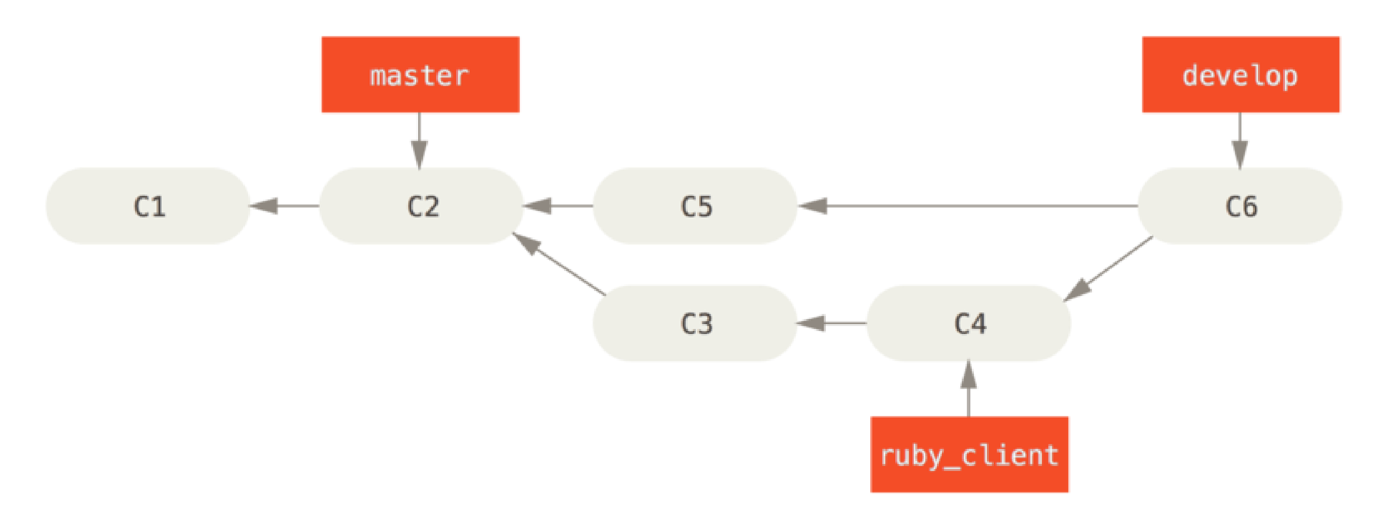


Figure 76. Tras integrar una rama puntual.

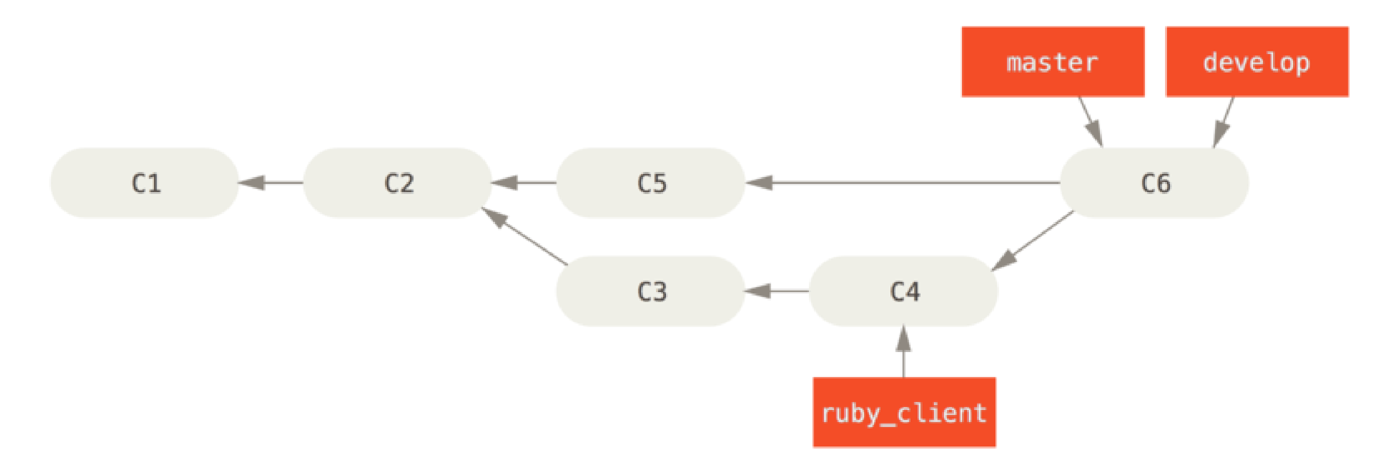


Figure 77. Tras el lanzamiento de una rama puntual.

De esta forma, cuando alguien clone el repositorio de tu proyecto, puede recuperar la rama `master` para construir la última versión estable y mantenerla actualizada fácilmente, o bien puede recuperar la rama `develop`, que es la que tiene los últimos desarrollos. Puedes ir un paso más allá y crear una rama de integración `integrate`, donde integres todo el trabajo. Entonces, cuando el código

de esa rama sea estable y pase las pruebas, lo puedes integrar en una rama de desarrollo; y cuando se demuestre que efectivamente permanece estable durante un tiempo, avanzas la rama `master`.

Flujos de trabajo con grandes integraciones

El proyecto Git tiene cuatro ramas de largo recorrido: `master`, `next`, y `pu` (proposed updates, actualizaciones propuestas) para trabajos nuevos, y `maint` para trabajos de mantenimiento de versiones anteriores. Cuando los colaboradores introducen nuevos trabajos, se recopilan en ramas puntuales en el repositorio del responsable de mantenimiento, de manera similar a la que se ha descrito (ver Gestionando un conjunto complejo de ramas puntuales paralelas.). En este punto, los nuevos trabajos se evalúan para decidir si son seguros y si están listos para los usuarios o si por el contrario necesitan más trabajo. Si son seguros, se integran en la rama `next`, y se envía dicha rama al repositorio público para que todo el mundo pueda probar las nuevas funcionalidades ya integradas.

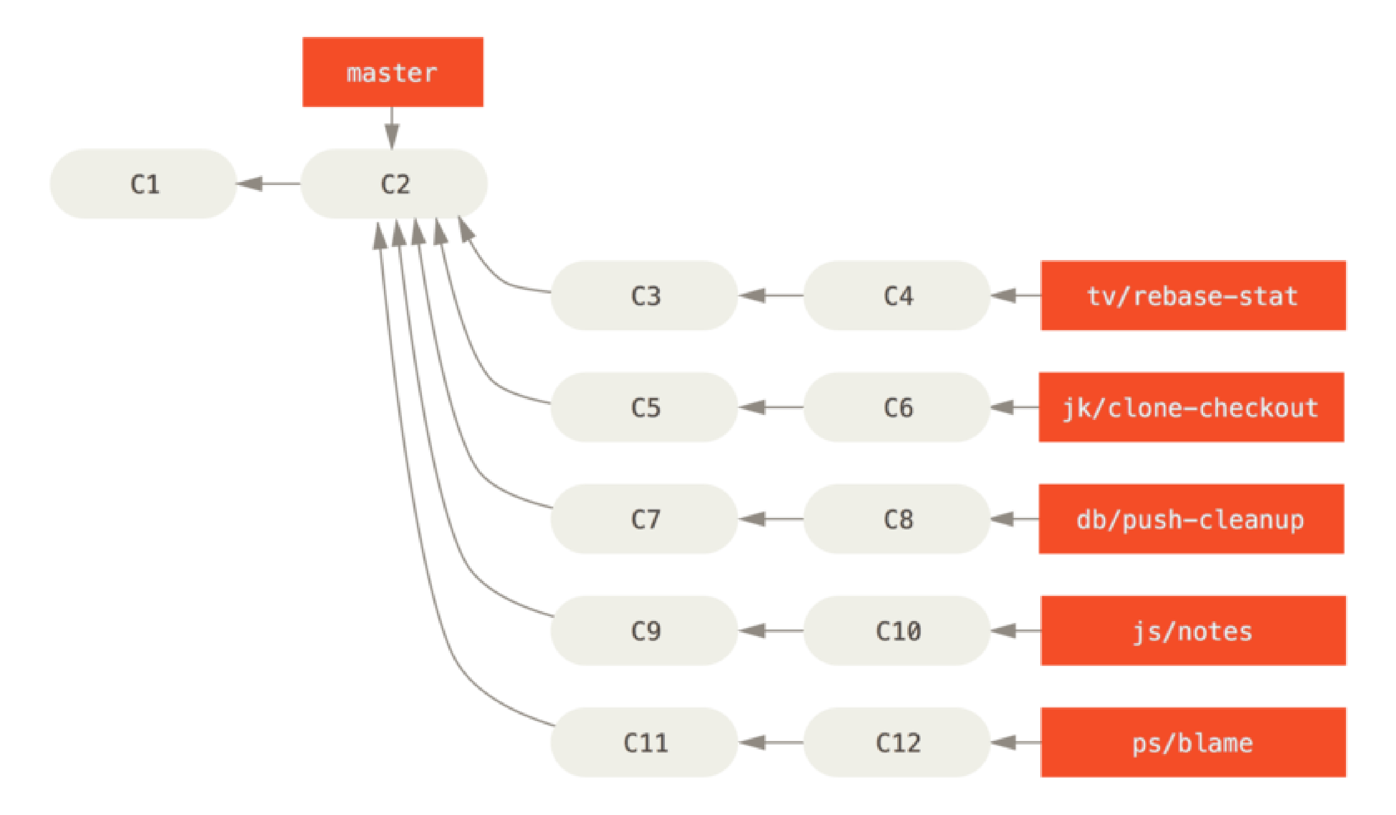


Figure 78. Gestionando un conjunto complejo de ramas puntuales paralelas.

Si las nuevas funcionalidades necesitan más trabajo, se integran en la rama `pu`. Cuando se decide que estas funcionalidades son totalmente estables, se integran de nuevo en la rama `master`, construyéndolas a partir de las funcionalidades en la rama `next` que aún no habían pasado a la rama `master`. Esto significa que la rama `master` casi siempre avanza, `next` se reorganiza ocasionalmente y `pu` se reorganiza mucho más a menudo:

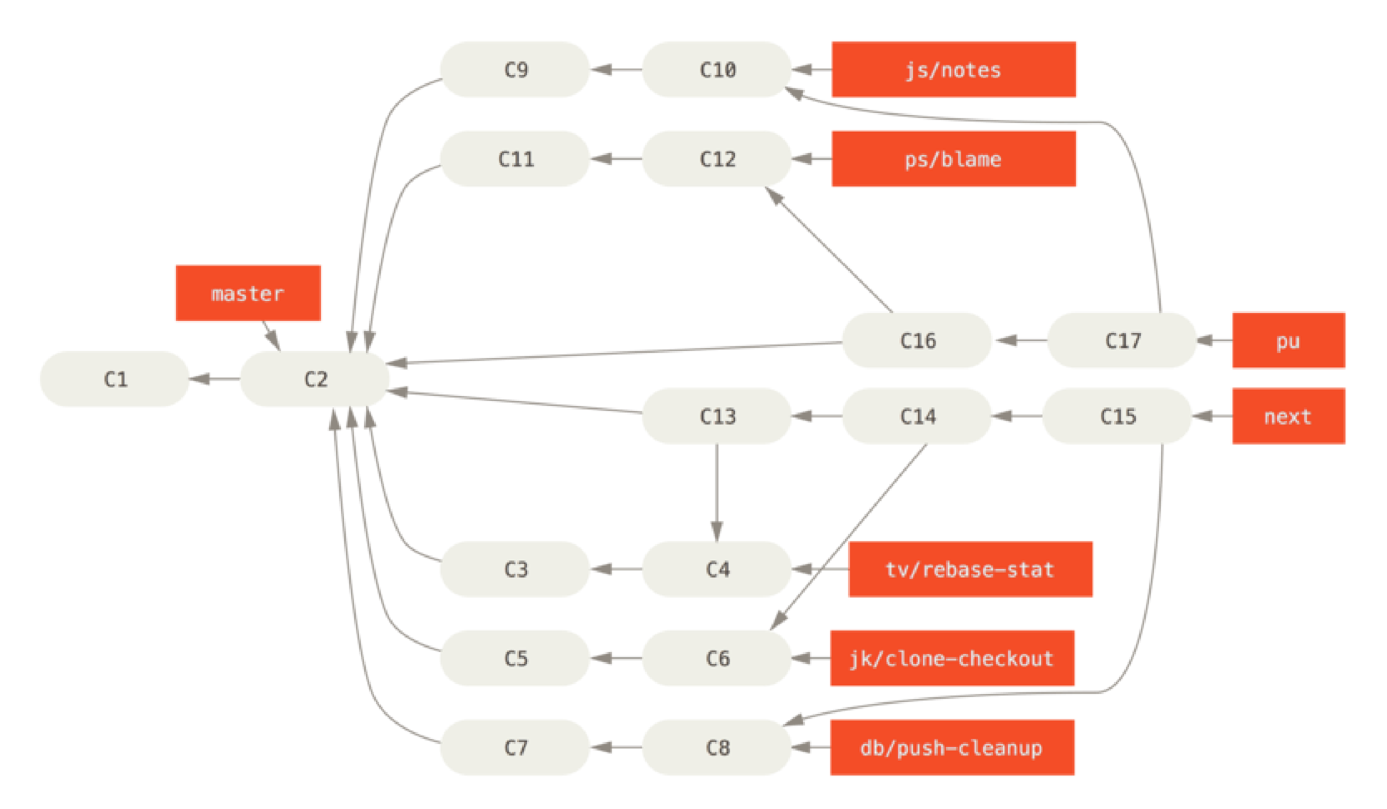


Figure 79. Fusionando ramas puntuales en ramas de integración de largo recorrido.

Cuando una rama puntual se ha integrado en la rama `master`, se elimina del repositorio. El proyecto Git también tiene una rama `maint` creada a partir de la última versión para ofrecer parches, en caso de que fuera necesaria una versión de mantenimiento. Así, cuando clonas el repositorio de Git, tienes cuatro ramas que puedes recuperar para evaluar el proyecto en diferentes etapas de desarrollo, dependiendo de si quieres tener una versión muy avanzada o de cómo quieras contribuir. De esta forma, el responsable de mantenimiento tiene un flujo de trabajo estructurado para ayudarle a aprobar las nuevas contribuciones.

Flujos de trabajo reorganizando o entresacando

Otros responsables de mantenimiento prefieren reorganizar o entresacar el nuevo trabajo en su propia rama `master`, en lugar de integrarlo, para mantener un historial prácticamente lineal. Cuando tienes trabajo en una rama puntual y has decidido que quieres integrarlo, te posicionas en esa rama y ejecutas el comando `rebase` para reconstruir los cambios en tu rama `master` (o `develop`, y así sucesivamente). Si ese proceso funciona bien, puedes avanzar tu rama `master`, consiguiendo un historial lineal en tu proyecto.

Otra forma de mover trabajo de una rama a otra es entresacarlo (cherry-pick). En Git, "entresacar" es como hacer un `rebase` para un único commit. Toma el parche introducido en un commit e intenta reaplicarlo en la rama en la que estás actualmente. Esto es útil si tienes varios commits en una rama puntual y sólo quieres integrar uno de ellos, o si sólo tienes un commit en una rama puntual y prefieres entresacarlo en lugar de hacer una reorganización (rebase). Por ejemplo, imagina que tienes un proyecto como éste:

Figure 80. Ejemplo de historial, antes de entresacar.

Si sólo deseas integrar el commit `e43a6` en tu rama `master`, puedes ejecutar

```
$ git cherry-pick e43a6fd3e94888d76779ad79fb568ed180e5fcdf
Finished one cherry-pick.
[master]: created a0a41a9: "More friendly message when locking the index fails."
 3 files changed, 17 insertions(+), 3 deletions(-)
```

Esto introduce el mismo cambio introducido en `e43a6`, pero genera un nuevo valor SHA-1 de confirmación, ya que la fecha en que se ha aplicado es distinta. Ahora tu historial queda así:

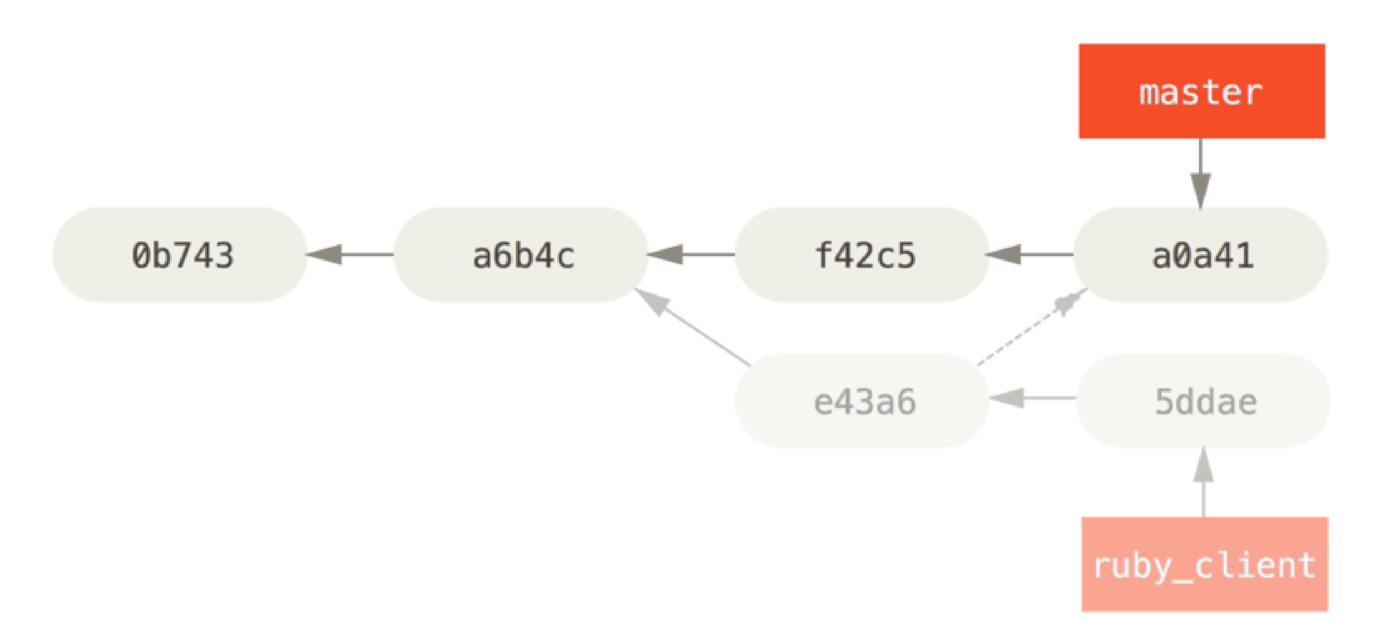


Figure 81. Historial tras entresacar un commit de una rama puntual.

En este momento ya puedes eliminar tu rama puntual y descartar los commits que no quieres integrar.

Rerere

Git dispone de una utilidad llamada “rerere” que puede resultar útil si estás haciendo muchas integraciones y reorganizaciones, o si mantienes una rama puntual de largo recorrido.

Rerere significa “reuse recorded resolution” (reutilizar resolución grabada) – es una forma de simplificar la resolución de conflictos. Cuando rerere está activo, Git mantendrá un conjunto de imágenes anteriores y posteriores a las integraciones correctas, de forma que si detecta que hay un conflicto que parece exactamente igual que otro ya corregido previamente, usará esa misma corrección sin causarte molestias.

Esta funcionalidad consta de dos partes: un parámetro de configuración y un comando. El parámetro de configuración es `rerere.enabled` y es bastante útil ponerlo en tu configuración global:

```
$ git config --global rerere.enabled true
```

Ahora, cuando hagas una integración que resuelva conflictos, la resolución se grabará en la caché por si la necesitas en un futuro.

Si fuera necesario, puedes interactuar con la cache de rerere usando el comando `git rerere`. Cuando se invoca sin ningún parámetro adicional, Git comprueba su base de datos de resoluciones en busca de coincidencias con cualquier conflicto durante la integración actual e intenta resolverlo (aunque se hace automáticamente en caso de que `rerere.enabled` sea `true`). También existen subcomandos para ver qué se grabará, para eliminar de la caché una resolución específica y para limpiar completamante la caché. Veremos más detalles sobre rerere en Rerere.

Etiquetando tus versiones

Cuando decides lanzar una versión, probablemente quieras etiquetarla para poder generarla más adelante en cualquier momento. Puedes crear una nueva etiqueta siguiendo los pasos descritos en Fundamentos de Git. Si decides firmar la etiqueta como responsable de mantenimiento, el etiquetado sería algo así:

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'
You need a passphrase to unlock the secret key for
user: "Scott Chacon <schacon@gmail.com>"
1024-bit DSA key, ID F721C45A, created 2009-02-09
```

Si firmas tus etiquetas podrías tener problemas a la hora de distribuir la clave PGP pública usada para firmarlas. El responsable de mantenimiento del proyecto Git ha conseguido solucionar este problema incluyendo su clave pública como un objeto binario en el repositorio, añadiendo a continuación una etiqueta que apunta directamente a dicho contenido. Para hacer esto, puedes averiguar qué clave necesitas lanzando el comando `gpg --list-keys`:

```
$ gpg --list-keys
/Users/schacon/.gnupg/pubring.gpg
---------------------------------
pub   1024D/F721C45A 2009-02-09 [expires: 2010-02-09]
uid                  Scott Chacon <schacon@gmail.com>
sub   2048g/45D02282 2009-02-09 [expires: 2010-02-09]
```

Ahora ya puedes importar directamente la clave en la base de datos de Git, exportándola y redirigiéndola a través del comando `git hash-object`, que escribe en Git un nuevo objeto binario con esos contenidos y devuelve la firma SHA-1 de dicho objeto.

```
$ gpg -a --export F721C45A | git hash-object -w --stdin
659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Una vez que tienes los contenidos de tu clave en Git, puedes crear una etiqueta que apunte directamente a dicha clave indicando el nuevo valor SHA-1 que devolvió el comando `hash-object`:

```
$ git tag -a maintainer-pgp-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Si ejecutas `git push --tags`, la etiqueta `maintainer-pgp-pub` será compartida con todo el mundo. Si

alguien quisiera verificar una etiqueta, puede importar tu clave PGP recuperando directamente el objeto binario de la base de datos e importándolo en GPG:

```
$ git show maintainer-pgp-pub | gpg --import
```

Esa clave se puede usar para verificar todas tus etiquetas firmadas. Además, si incluyes instrucciones en el mensaje de la etiqueta, el comando `git show <tag>` permitirá que el usuario final obtenga instrucciones más específicas sobre el proceso de verificación de etiquetas.

Generando un número de compilación

Como Git no genera una serie de números monótonamente creciente como *v123* o similar con cada commit, si quieres tener un nombre más comprensible para un commit, puedes ejecutar el comando `git describe` sobre dicho commit. Git devolverá el nombre de la etiqueta más próxima junto con el número de commits sobre esa etiqueta y una parte del valor SHA-1 del commit que estás describiendo:

```
$ git describe master
v1.6.2-rc1-20-g8c5b85c
```

De esta forma, puedes exportar una instantánea o generar un nombre comprensible por cualquier persona. De hecho, si construyes Git a partir del código fuente clonado del repositorio Git, `git --version` devuelve algo parecido a esto. Si estás describiendo un commit que has etiquetado directamente, te dará el nombre de la etiqueta.

El comando `git describe` da preferencia a etiquetas anotadas (etiquetas creadas con las opciones `-a` o `-s`), por lo que las etiquetas de versión deberían crearse de esta forma si estás usando `git describe`, para garantizar que el commit es nombrado adecuadamente cuando se describe. También puedes usar esta descripción como objetivo de un comando `checkout` o `show`, aunque depende de la parte final del valor SHA-1 abreviado, por lo que podría no ser válida para siempre. Por ejemplo, recientemente el núcleo de Linux pasó de 8 a 10 caracteres para asegurar la unicidad del objeto SHA-1, por lo que los nombres antiguos devueltos por `git describe` fueron invalidados.

Preparando una versión

Ahora quieres lanzar una versión. Una cosa que querrás hacer será crear un archivo con la última instantánea del código para esas pobres almas que no usan Git. El comando para hacerlo es `git archive`:

```
$ git archive master --prefix='project/' | gzip > `git describe master`.tar.gz
$ ls *.tar.gz
v1.6.2-rc1-20-g8c5b85c.tar.gz
```

Si alguien abre el archivo tar, obtiene la última instantánea de tu proyecto bajo un directorio `project`. También puedes crear un archivo zip de la misma manera, pero añadiendo la opción `--format=zip` a `git archive`:

```
$ git archive master --prefix='project/' --format=zip > `git describe master`.zip
```

Ahora tienes tanto un archivo tar como zip con la nueva versión de tu proyecto, listos para subirlos a tu sitio web o para enviarlos por e-mail.

El registro resumido

Es el momento de enviar un mensaje a tu lista de correo informando sobre el estado de tu proyecto. Una buena opción para obtener rápidamente una especie de lista con los cambios introducidos en tu proyecto desde la última versión o e-mail es usar el comando `git shortlog`. Dicho comando resume todos los commits en el rango que se le indique; por ejemplo, el siguiente comando devuelve un resumen de todos los commits desde tu última versión, suponiendo que fuera la v1.0.1:

```
$ git shortlog --no-merges master --not v1.0.1
Chris Wanstrath (8):
      Add support for annotated tags to Grit::Tag
      Add packed-refs annotated tag support.
      Add Grit::Commit#to_patch
      Update version and History.txt
      Remove stray `puts`
      Make ls_tree ignore nils

Tom Preston-Werner (4):
      fix dates in history
      dynamic version method
      Version bump to 1.0.2
      Regenerated gemspec for version 1.0.2
```

Consigues un resumen limpio de todos los commits desde la versión v1.0.1, agrupados por autor, que puedes enviar por correo electrónico a tu lista.

Resumen

Deberías sentirte lo suficiente cómodo para contribuir en un proyecto en Git así como para mantener tu propio proyecto o integrar las contribuciones de otros usuarios. ¡Felicidades por ser un desarrollador eficaz con Git! En el próximo capítulo, aprenderás como usar el servicio más grande y popular para alojar proyectos de Git: Github.

GitHub

GitHub es el mayor proveedor de alojamiento de repositorios Git, y es el punto de encuentro para que millones de desarrolladores colaboren en el desarrollo de sus proyectos. Un gran porcentaje de los repositorios Git se almacenan en GitHub, y muchos proyectos de código abierto lo utilizan para hospedar su Git, realizar su seguimiento de fallos, hacer revisiones de código y otras cosas. Por tanto, aunque no sea parte directa del proyecto de código abierto de Git, es muy probable que durante tu uso profesional de Git necesites interactuar con GitHub en algún momento.

Este capítulo trata del uso eficaz de GitHub. Veremos cómo crear y gestionar una cuenta, crear y gestionar repositorios Git, también los flujos de trabajo (workflows) habituales para participar en proyectos y para aceptar nuevos participantes en los tuyos, la interfaz de programación de GitHub (API) y muchos otros pequeños trucos que te harán, en general, la vida más fácil.

Si no vas a utilizar GitHub para hospedar tus proyectos o para colaborar con otros, puedes saltar directamente a Herramientas de Git.

WARNING

Cambios en la interfaz

Observa que como muchos sitios web activos, el aspecto de la interfaz de usuario puede cambiar con el tiempo, frente a las capturas de pantalla que incluye este libro. Probablemente la versión en línea de este libro tenga esas capturas más actualizadas.

Creación y configuración de la cuenta

Lo primero que necesitas es una cuenta de usuario gratuita. Simplemente visita https://github.com, elige un nombre de usuario que no esté ya en uso, proporciona un correo y una contraseña, y pulsa el botón verde grande “Sign up for GitHub”.

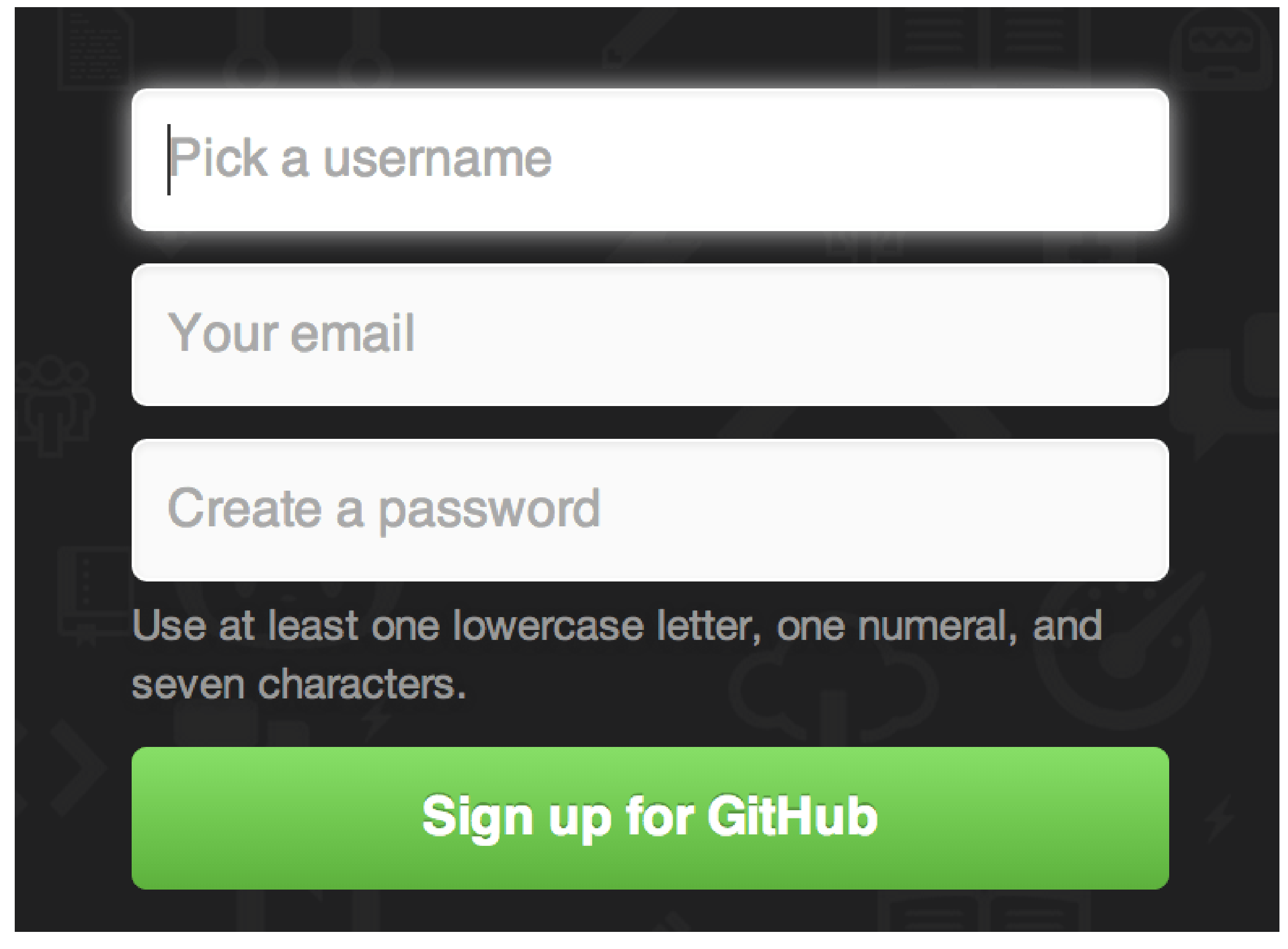


Figure 82. Formulario para darse de alta en GitHub.

Lo siguiente que verás es la página de precios para planes mejores, pero lo puedes ignorar por el momento. GitHub te enviará un correo para verificar la dirección que les has dado. Confirma la dirección ahora, es bastante importante (como veremos después).

NOTE	GitHub proporciona toda su funcionalidad en cuentas gratuitas, con la limitación de que todos tus proyectos serán públicos (todos los usuarios tendrán acceso de lectura). Los planes de pago de GitHub te permite tener algunos proyectos privados, pero esto es algo que no veremos en este libro.

Si pulsas en el logo del gato con patas de pulpo en la parte superior izquierda de la pantalla llegarás a tu escritorio principal. Ahora ya estás listo para comenzar a usar GitHub.

Acceso SSH

Desde ya, puedes acceder a los repositorios Git utilizando el protocolo `https://`, identificándote con el usuario y la contraseña que acabas de elegir. Sin embargo, para simplificar el clonado de proyectos públicos, no necesitas crearte la cuenta. Es decir, la cuenta solo la necesitas cuando comienzas a hacer cosas como bifurcar (fork) proyectos y enviar tus propios cambios más tarde.

Si prefieres usar SSH, necesitas configurar una clave pública. Si aun no la tienes, mira cómo generarla en Generando tu clave pública SSH.) Abre tu panel de control de la cuenta utilizando el enlace de la parte superior derecha de la ventana:

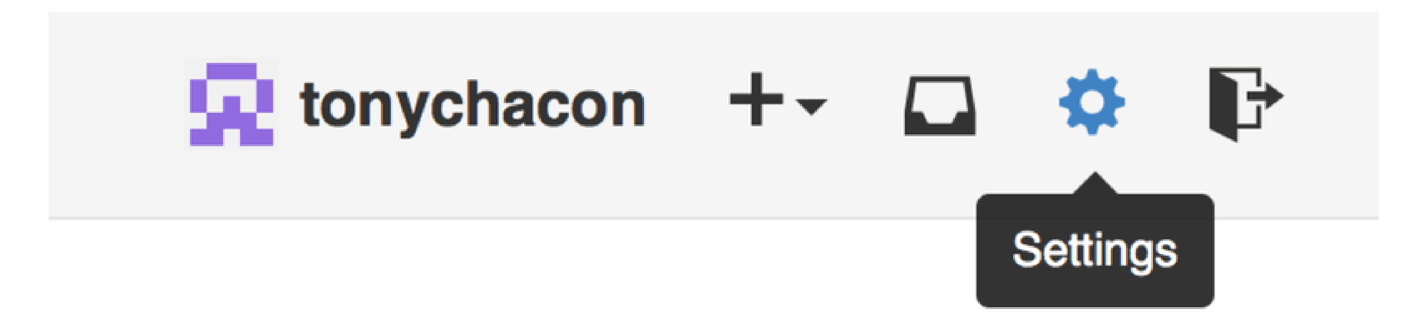


Figure 83. Enlace “Account settings”.

Aquí selecciona en el lado izquierdo la opción “SSH keys”.

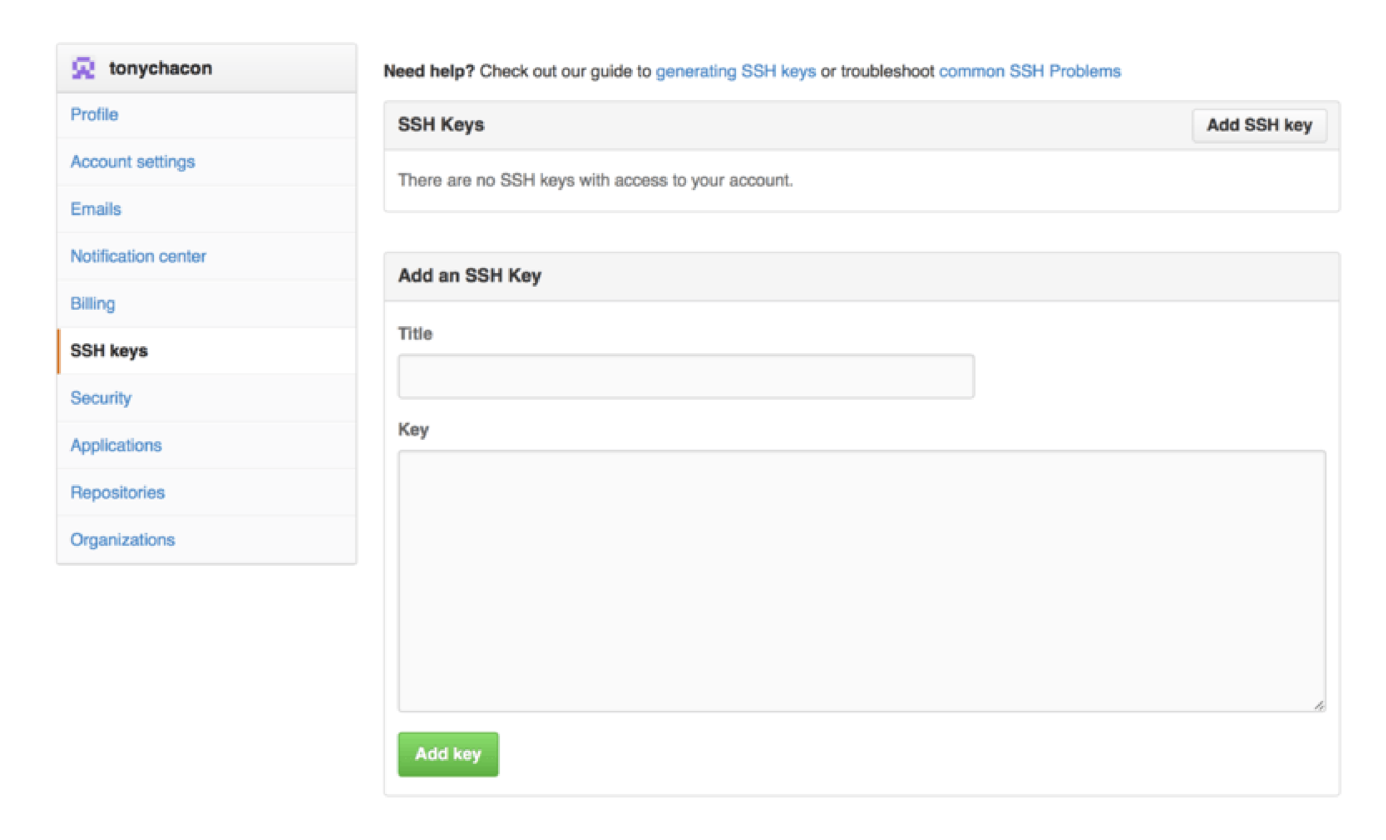


Figure 84. Enlace “SSH keys”.

Desde ahí, pulsa sobre "`Add an SSH key`", proporcionando un nombre y pegando los contenidos del archivo `~/.ssh/id_rsa.pub` (o donde hayas definido tu clave pública) en el área de texto, y pulsa sobre “Add key”.

NOTE

Asegúrate darle a tu clave un nombre que puedas recordar. Puedes, por ejemplo, añadir claves diferentes, con nombres como "Clave Portátil" o "Cuenta de trabajo" de modo que si tienes que revocar alguna clave más tarde, te resultará más fácil decidir cuál es.

Tu icono

También, si quieres, puedes reemplazar el icono (avatar) que te generaron para ti con una imagen de tu elección. En primer lugar selecciona la opción “Profile” (encima de la opción de “SSH keys”) y pulsa sobre “Upload new picture”.

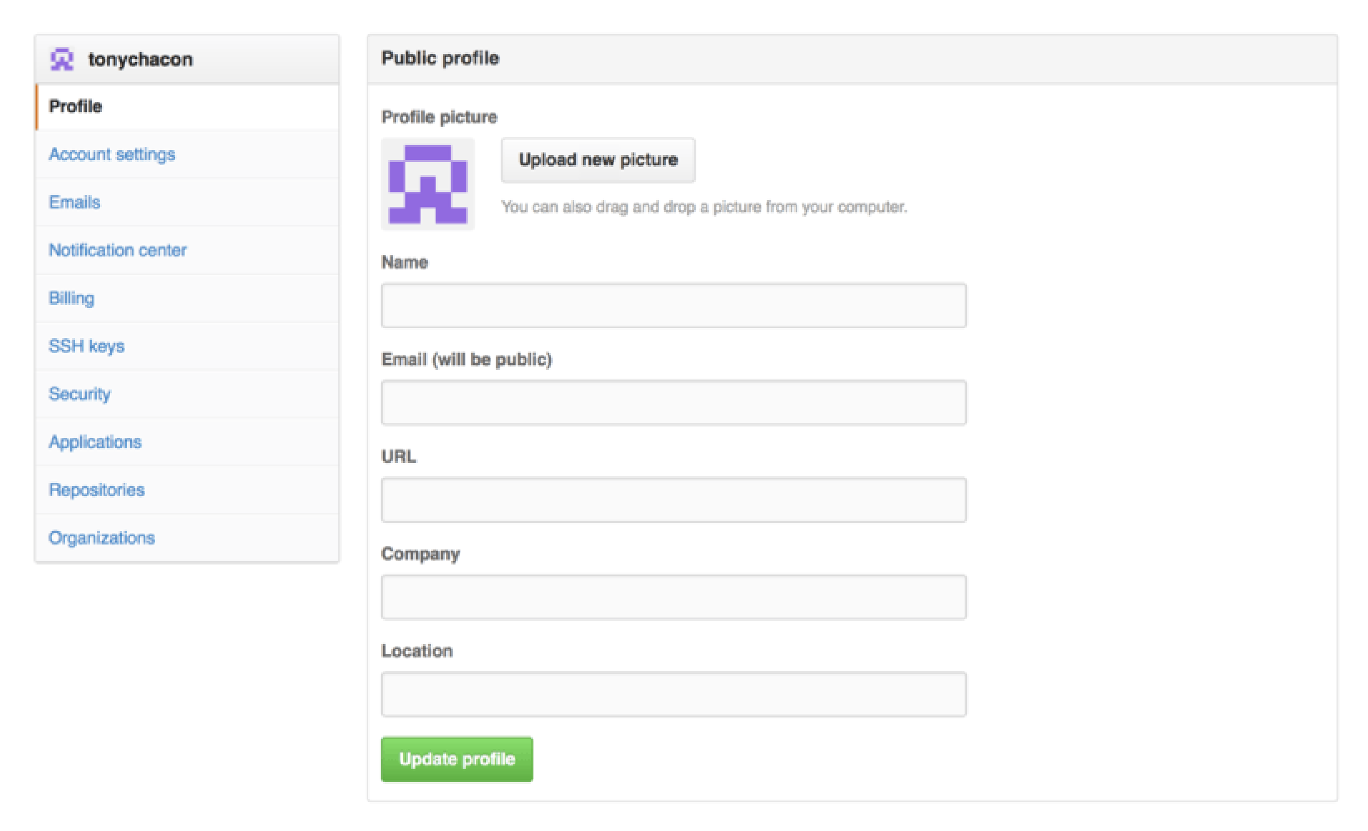


Figure 85. Enlace "Profile".

Nosotros eligiremos como ejemplo una copia del logo de Git que tengamos en el disco duro y luego tendremos opción de recortarlo al subirlo.

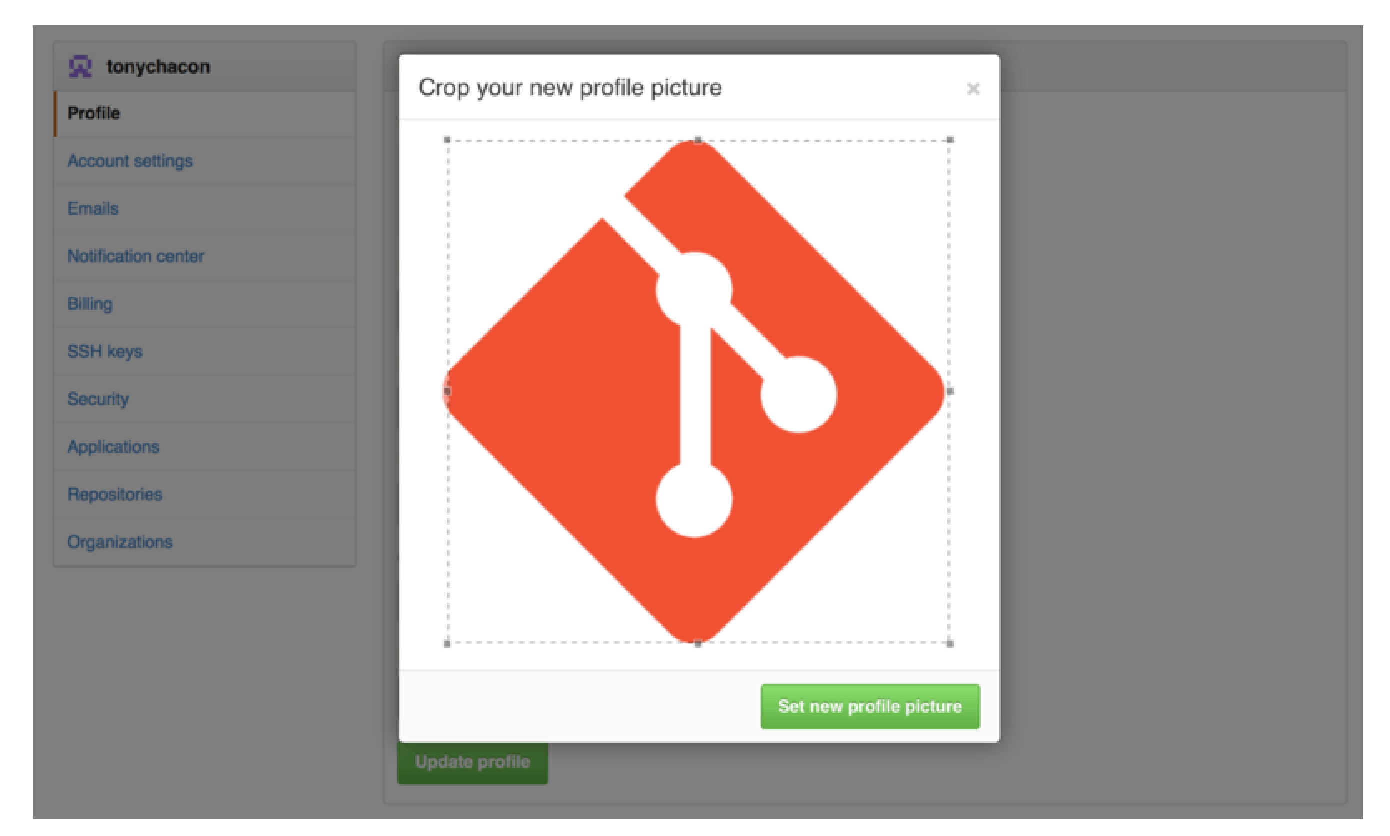


Figure 86. Recortar tu icono

Desde ahora, quien vea tu perfil o tus contribuciones a repositorios verá tu nuevo icono junto a tu nombre.

Si da la casualidad que ya tienes tu icono en el popular servicio Gravatar (conocido por su uso en las cuentas de Wordpress), este icono será detectado y no tendrás que hacer, si quieres, este paso.

Tus direcciones de correo

La forma con la que GitHub identifica tus contribuciones a Git es mediante la dirección de correo electrónico. Si tienes varias direcciones diferentes en tus contribuciones (commits) y quieres que GitHub sepa que son de tu cuenta, necesitas añadirlas todas en la sección Emails de la sección de administración.

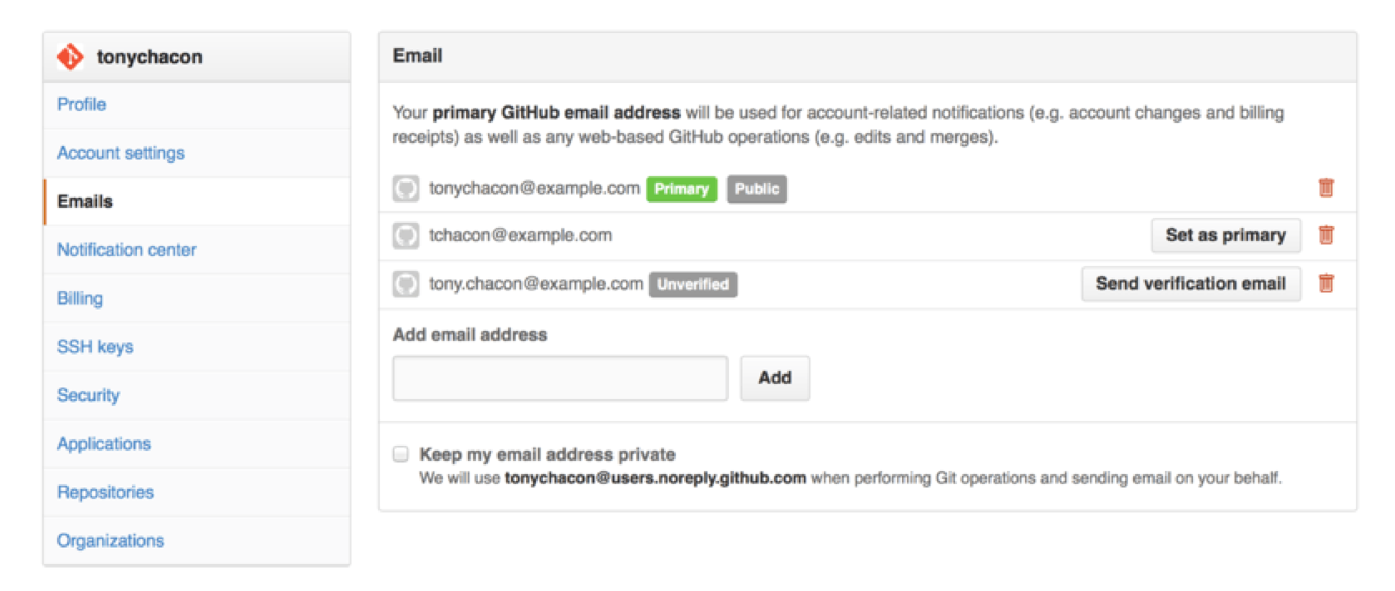


Figure 87. Añadiendo direcciones de correo

En Añadiendo direcciones de correo podemos ver los diferentes estados posibles. La dirección inicial se verifica y se utiliza como dirección principal, lo que significa que es donde vas a recibir cualquier notificación. La siguiente dirección se puede verificar y ponerla entonces como dirección principal, si quieres cambiarla. La última dirección no está verificada, lo que significa que no puedes usarla como principal. Pero si GitHub ve un commit con esa dirección, la identificará asociándola a tu usuario.

Autentificación de dos pasos

Finalmente, y para mayor seguridad, deberías configurar la Autentificación de Dos Pasos o “2FA”. Este tipo de autentificación se está haciendo más popular para reducir el riesgo de que te roben la cuenta. Al activarla, GitHub te pedirá identificarte de dos formas, de forma que si una de ellas resulta comprometida, el atacante no conseguirá acceso a tu cuenta.

Puedes encontrar la configuración de “2FA” en la opción Security de los ajustes de la cuenta.

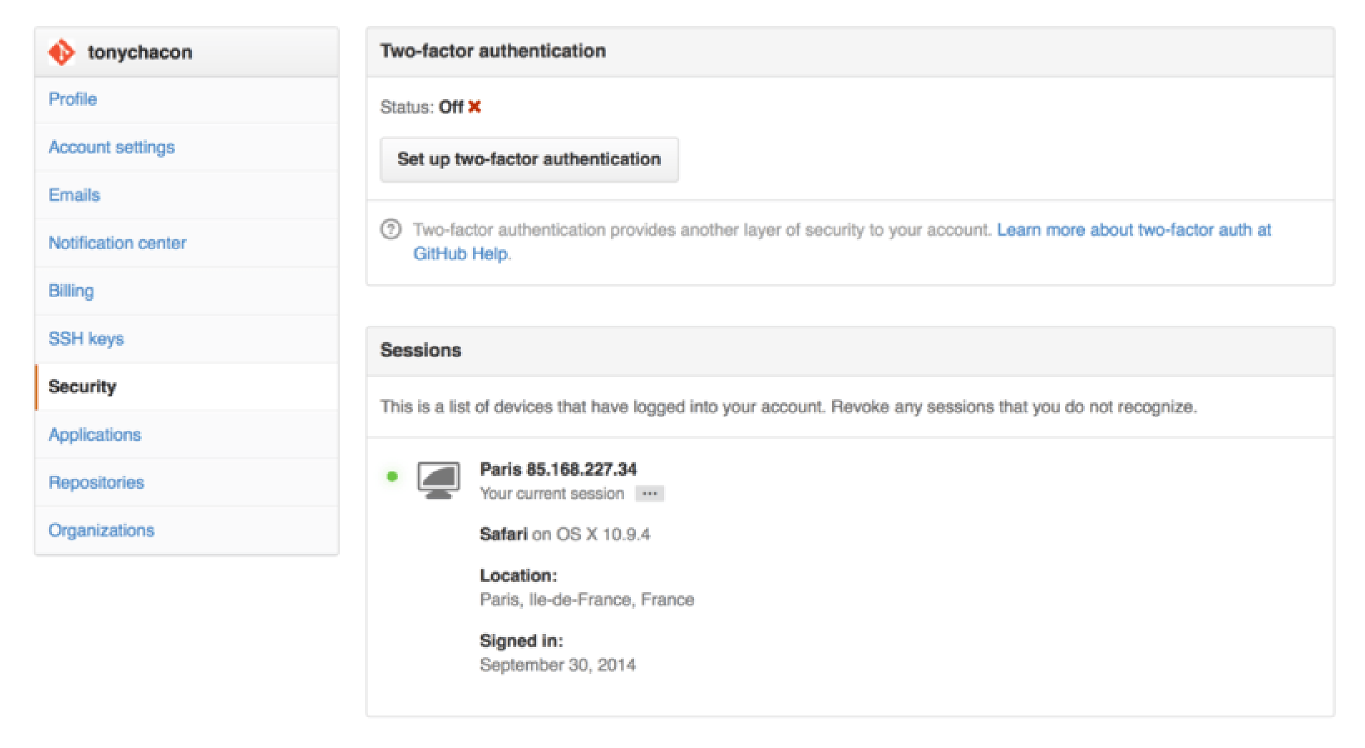


Figure 88. 2FA dentro de Security

Si pulsas en el botón “Set up two-factor authentication”, te saldrá una página de configuración donde podrás elegir un generador de códigos en una aplicación de móvil (es decir, códigos de un solo uso) o bien podrás elegir que te envíen un SMS cada vez que necesites entrar.

Cuando configures este método de autentificación, tu cuenta será un poco más segura ya que tendrás que proporcionar un código junto a tu contraseña cada vez que accedas a GitHub.

Participando en Proyectos

Una vez que tienes la cuenta configurada, veremos algunos detalles útiles para ayudarte a participar en proyectos existentes.

Bifurcación (fork) de proyectos

Si quieres participar en un proyecto existente, en el que no tengas permisos de escritura, puedes bifurcarlo (hacer un “fork”). Esto consiste en crear una copia completa del repositorio totalmente bajo tu control: se encontrará en tu cuenta y podrás escribir en él sin limitaciones.

NOTE	Históricamente, el término “fork” podía tener connotaciones algo negativas, ya que significaba que alguien realizaba una copia del código fuente del proyecto y las comenzaba a modificar de forma independiente al proyecto original, tal vez para crear un proyecto competidor y dividir a su comunidad de colaboradores. En GitHub, el “fork” es simplemente una copia del repositorio donde puedas escribir, haciendo públicos tus propios cambios, como una manera abierta de participación.

De esta forma, los proyectos no necesitan añadir colaboradores con acceso de escritura (push). La gente puede bifurcar un proyecto, enviar sus propios cambios a su copia y luego remitir esos cambios al repositorio original para su aprobación, creando lo que se llama un Pull Request, que veremos más adelante. Esto permite abrir una discusión para la revisión del código, donde

propietario y participante pueden comunicarse acerca de los cambios y, en última instancia, el propietario original puede aceptarlos e integrarlos en el proyecto original cuando lo considere adecuado.

Para bifurcar un proyecto, visita la página del mismo y pulsa sobre el botón “Fork” del lado superior derecho de la página.



Figure 89. Botón “Fork”.

En unos segundos te redireccionarán a una página nueva de proyecto, en tu cuenta y con tu propia copia del código fuente.

El Flujo de Trabajo en GitHub

GitHub está diseñado alrededor de un flujo de trabajo de colaboración específico, centrado en las solicitudes de integración (“pull request”). Este flujo es válido tanto si colaboras con un pequeño equipo en un repositorio compartido como si lo haces en una gran red de participantes con docenas de bifurcaciones particulares. Se centra en el workflow Ramas Puntuales cubierto en Ramificaciones en Git.

El funcionamiento habitual es el siguiente:

1. Se crea una rama a partir de `master`.
2. Se realizan algunos commits hacia esa rama.
3. Se envía esa rama hacia tu copia (fork) del proyecto.
4. Abres un Pull Request en GitHub.
5. Se participa en la discusión asociada y, opcionalmente, se realizan nuevos commits.
6. El propietario del proyecto original cierra el Pull Request, bien fusionando la rama con tus cambios o bien rechazándolos.

Este es, básicamente, el flujo de trabajo del Responsable de Integración visto en Integration-Manager Workflow, pero en lugar de usar el correo para comunicarnos y revisar los cambios, lo que se hace es usar las herramientas web de GitHub.

Veamos un ejemplo de cómo proponer un cambio en un proyecto de código abierto hospedado en GitHub, utilizando esta forma de trabajar.

Creación del Pull Request

Tony está buscando código para ejecutar en su microcontrolador Arduino, y ha encontrado un programa interesante en GitHub, en la dirección https://github.com/schacon/blink.

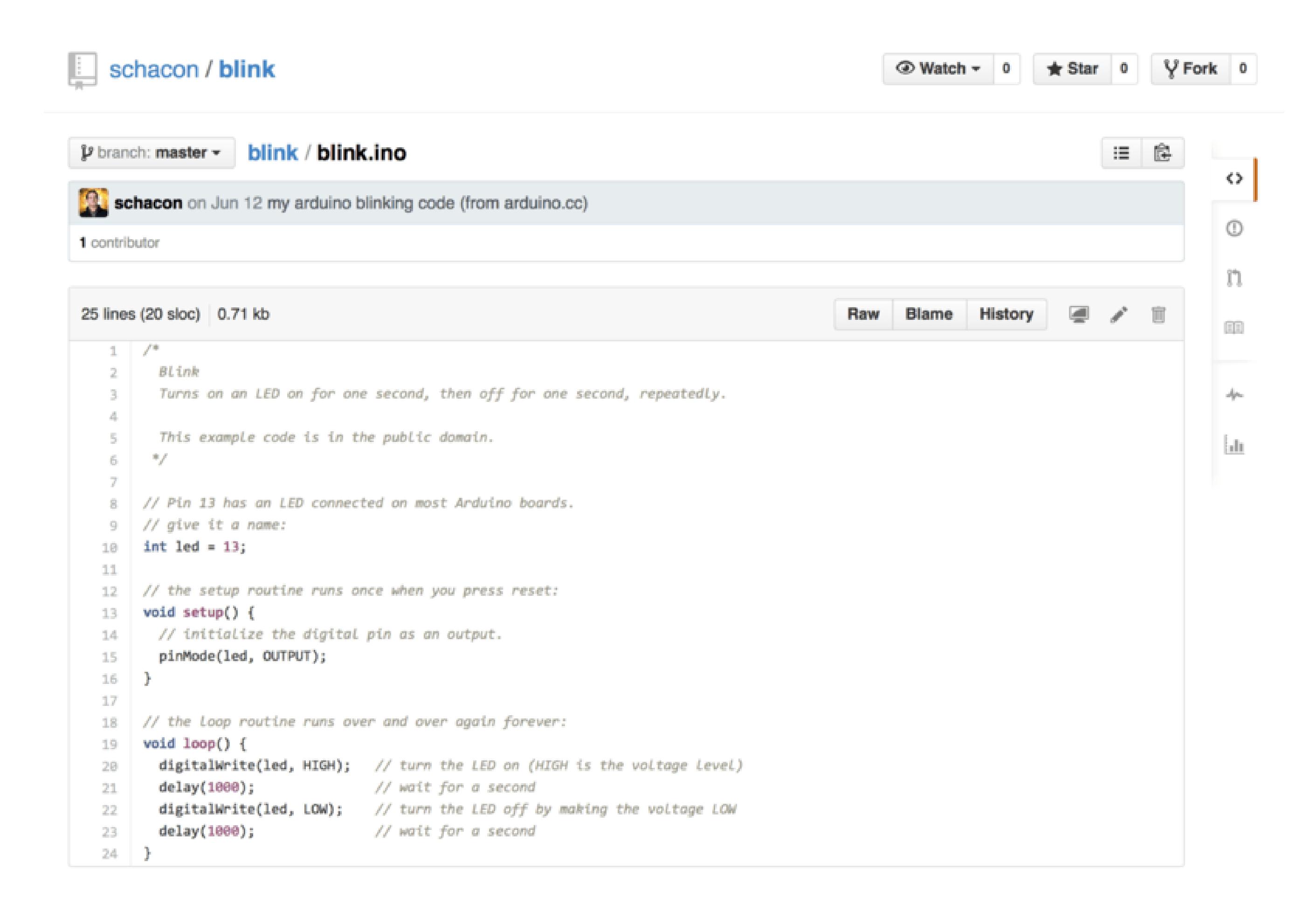


Figure 90. El proyecto en el que queremos participar.

El único problema es que la velocidad de parpadeo es muy rápida, y piensa que es mucho mejor esperar 3 segundos en lugar de 1 entre cada cambio de estado. Luego, nuestra mejora consistirá en cambiar la velocidad y enviar el cambio al proyecto como un cambio propuesto.

Lo primero que se hace, es pulsar en el botón *Fork* ya conocido para hacer nuestra propia copia del proyecto. Nuestro nombre de usuario es “tonychacon” por lo que la copia del proyecto tendrá como dirección `https://github.com/tonychacon/blink`, y en esta copia es donde podemos trabajar. La clonaremos localmente, crearemos una rama, realizaremos el cambio sobre el código fuente y finalmente enviaremos esos cambios a GitHub.

```
$ git clone https://github.com/tonychacon/blink ①
Cloning into 'blink'...

$ cd blink
$ git checkout -b slow-blink ②
Switched to a new branch 'slow-blink'

$ sed -i '' 's/1000/3000/' blink.ino ③

$ git diff --word-diff ④
diff --git a/blink.ino b/blink.ino
index 15b9911..a6cc5a5 100644
--- a/blink.ino
+++ b/blink.ino
@@ -18,7 +18,7 @@ void setup() {
// the loop routine runs over and over again forever:
void loop() {
  digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage level)
  [-delay(1000);-]{+delay(3000);+}               // wait for a second
  digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
  [-delay(1000);-]{+delay(3000);+}               // wait for a second
}

$ git commit -a -m 'three seconds is better' ⑤
[slow-blink 5ca509d] three seconds is better
 1 file changed, 2 insertions(+), 2 deletions(-)

$ git push origin slow-blink ⑥
Username for 'https://github.com': tonychacon
Password for 'https://tonychacon@github.com':
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 340 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/tonychacon/blink
 * [new branch]      slow-blink -> slow-blink
```

① Clonar nuestro fork en nuestro equipo

② Crear la rama, que sea descriptiva

③ Realizar nuestros cambios

④ Comprobar los cambios

⑤ Realizar un commit de los cambios en la rama

⑥ Enviar nuestra nueva rama de vuelta a nuestro fork

Ahora, si miramos nuestra bifurcación en GitHub, veremos que aparece un aviso de creación de la rama y nos dará la oportunidad de hacer una solicitud de integración con el proyecto original.

También puedes ir a la página “Branches” en `https://github.com/<user>/<project>/branches` para localizar la rama y abrir el Pull Request desde ahí.

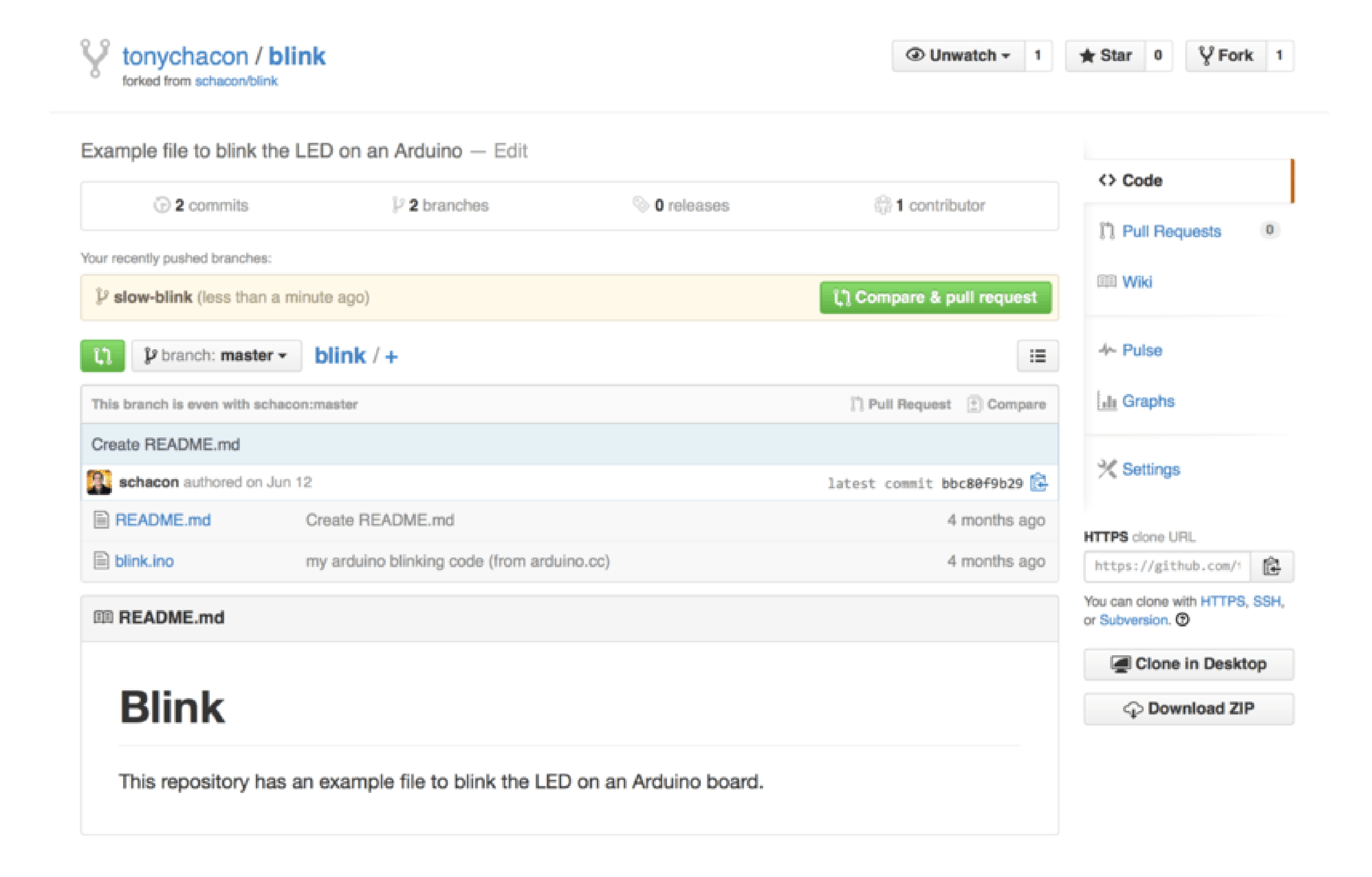


Figure 91. Botón Pull Request

Si pulsamos en el botón verde, veremos una pantalla que permite crear un título y una descripción para darle al propietario original una buena razón para tenerla en cuenta. Normalmente debemos realizar cierto esfuerzo en hacer una buena descripción para que el autor sepa realmente qué estamos aportando y lo valore adecuadamente.

También veremos la lista de commits de la rama que están “por delante” de la rama `master` (en este caso, la única) y un diff unificado de los cambios que se aplicarían si se fusionasen con el proyecto original.

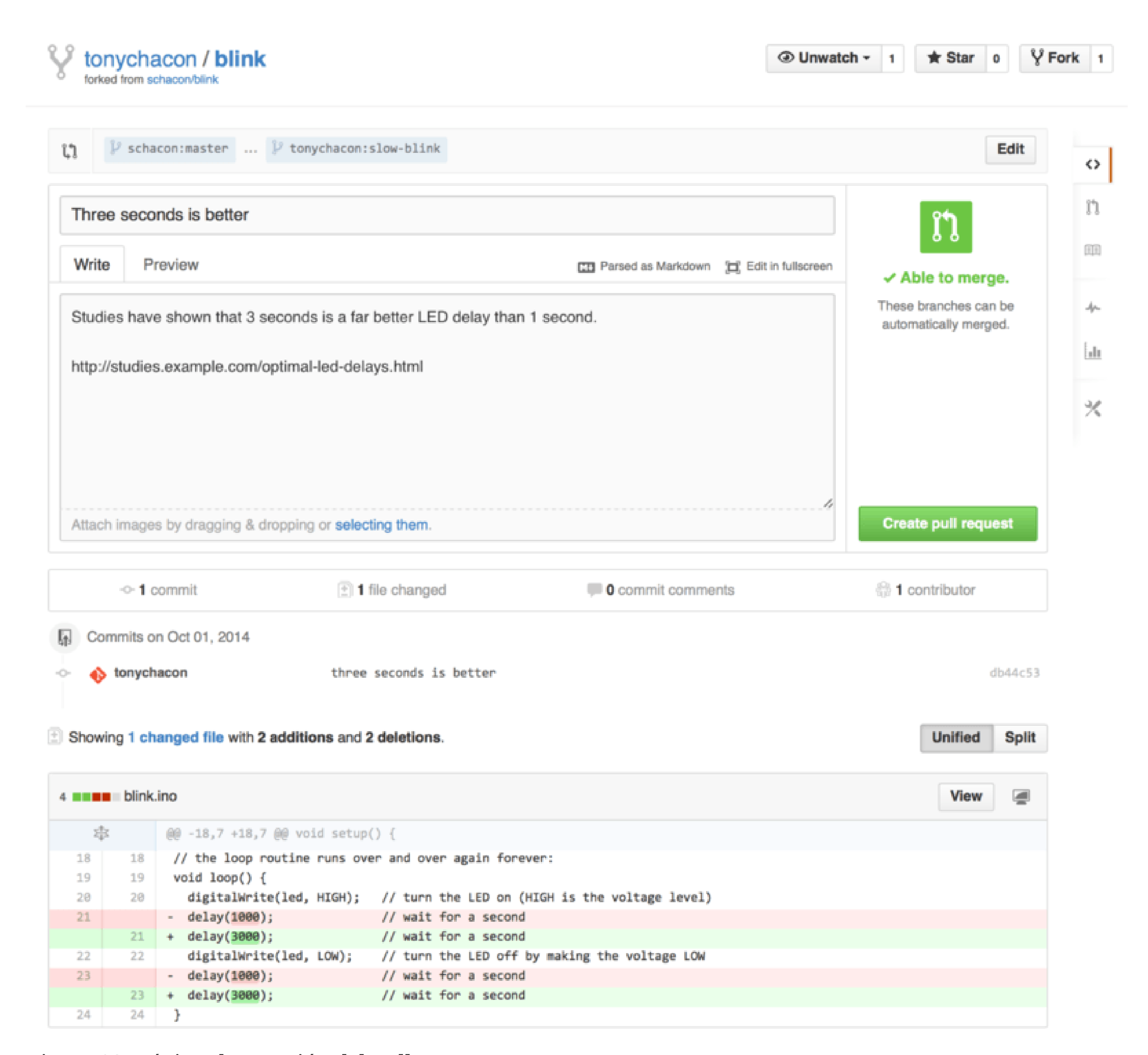


Figure 92. Página de creación del Pull Request

Cuando seleccionas el botón *Create pull request*, el propietario del proyecto que has bifurcado recibirá una notificación de que alguien sugiere un cambio junto a un enlace donde está toda la información.

> **NOTE**
>
> Aunque los Pull Request se utilizan en proyectos públicos como este donde el ayudante tiene un conjunto de cambios completos para enviar, también se utiliza en proyectos internos al principio del ciclo de desarrollo: puedes crear el Pull Request con una rama propia y seguir enviando commits a dicha rama después de crear el Pull Request, siguiendo un modelo iterativo de desarrollo, en lugar de crear la rama cuando ya has finalizado todo el trabajo.

Evolución del Pull Request

En este punto, el propietario puede revisar el cambio sugerido e incorporarlo (merge) al proyecto, o bien rechazarlo o comentarlo. Por ejemplo, si le gusta la idea pero prefiere esperar un poco.

La discusión, en los workflow de Git en entornos distribuidos, tiene lugar por correo electrónico,

mientras que en GitHub tiene lugar en línea. El propietario del proyecto puede revisar el diff y dejar un comentario pulsando en cualquier línea del diff.

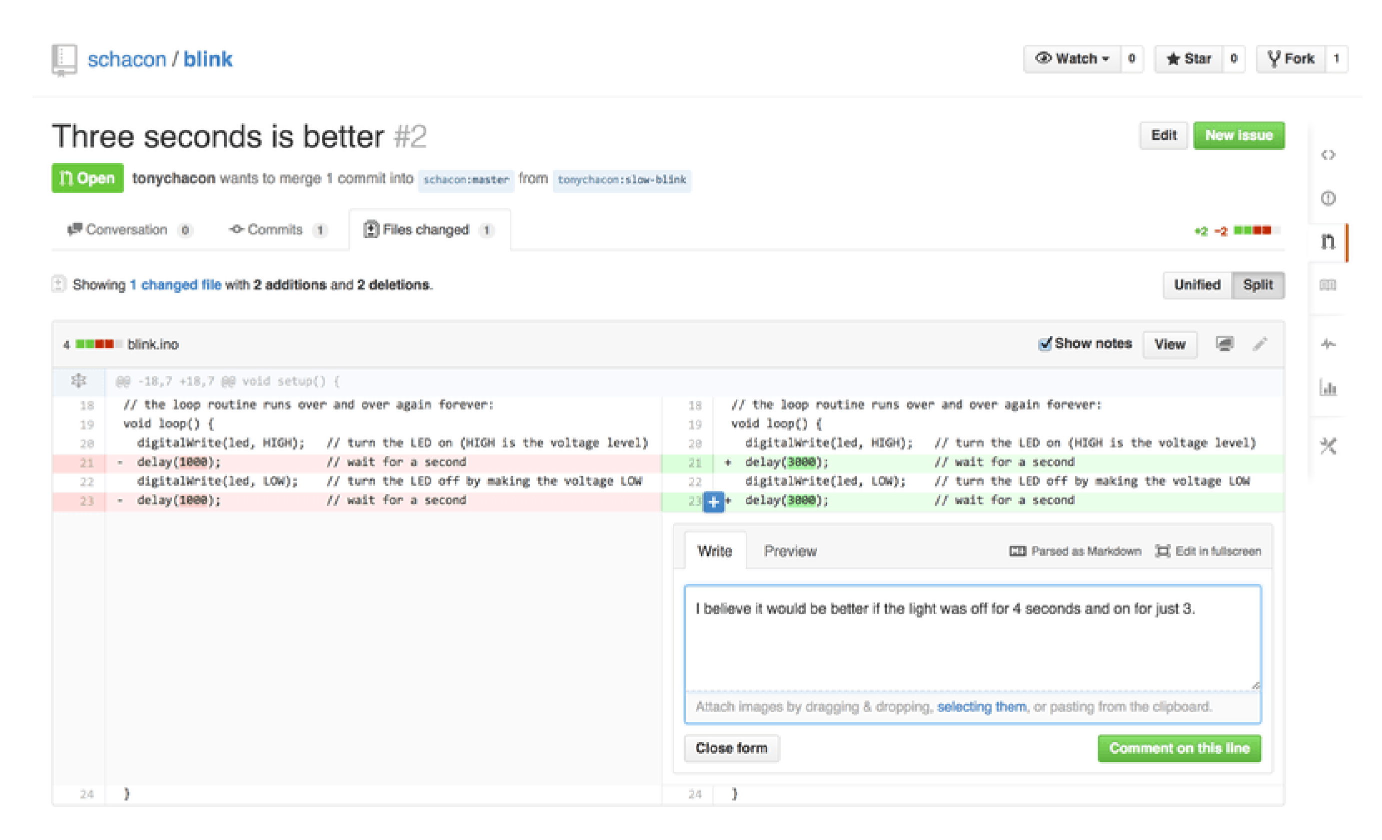


Figure 93. Comentando una línea concreta del diff

Cuando el responsable hace el comentario, la persona que solicitó la integración (y otras personas que hayan configurado sus cuentas para escuchar los cambios del repositorio) recibirán una notificación. Más tarde veremos cómo personalizar esto, pero si las notificaciones están activas, Tony recibiría un correo como este:



Figure 94. Comentarios enviados en notificaciones de correo

Cualquiera puede añadir sus propios comentarios. En Página de discusión del Pull Request vemos un ejemplo de propietario de proyecto comentando tanto una línea del código como dejando un comentario general en la sección de discusión. Puedes comprobar que los comentarios del código se insertan igualmente en la conversación.

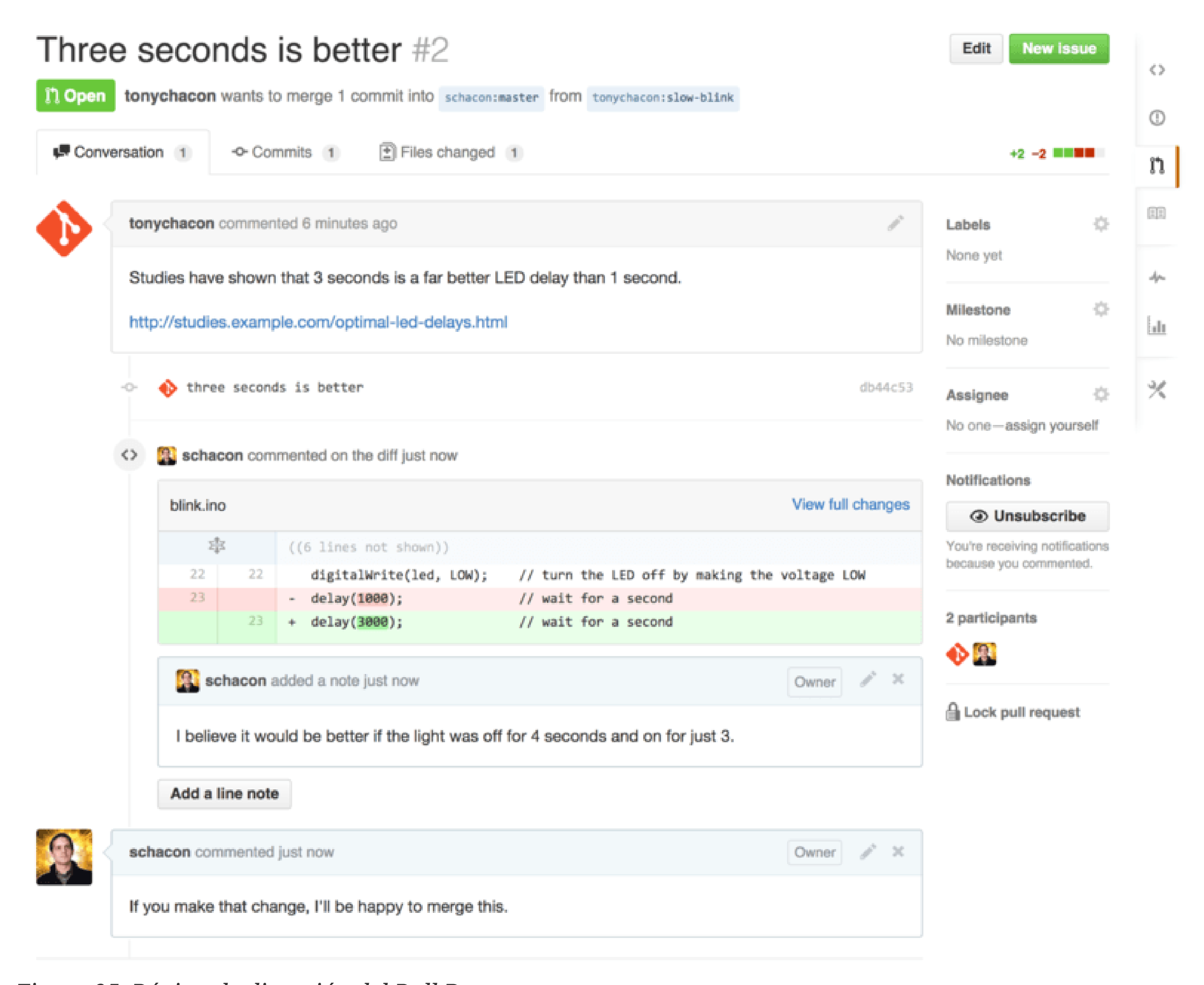


Figure 95. Página de discusión del Pull Request

El participante puede ver ahora qué tiene que hacer para ver aceptado su cambio. Con suerte sera poco trabajo. Mientras que con el correo electrónico tendrías que revisar los cambios y reenviarlos a la lista de correo, en GitHub puedes, simplemente, enviar un nuevo commit a la rama y subirla (push).

Si el participante hace esto, el coordinador del proyecto será notificado nuevamente y, cuando visiten la página, verán lo que ha cambiado. De hecho, al ver que un cambio en una línea de código tenía ya un comentario, GitHub se da cuenta y oculta el diff obsoleto.

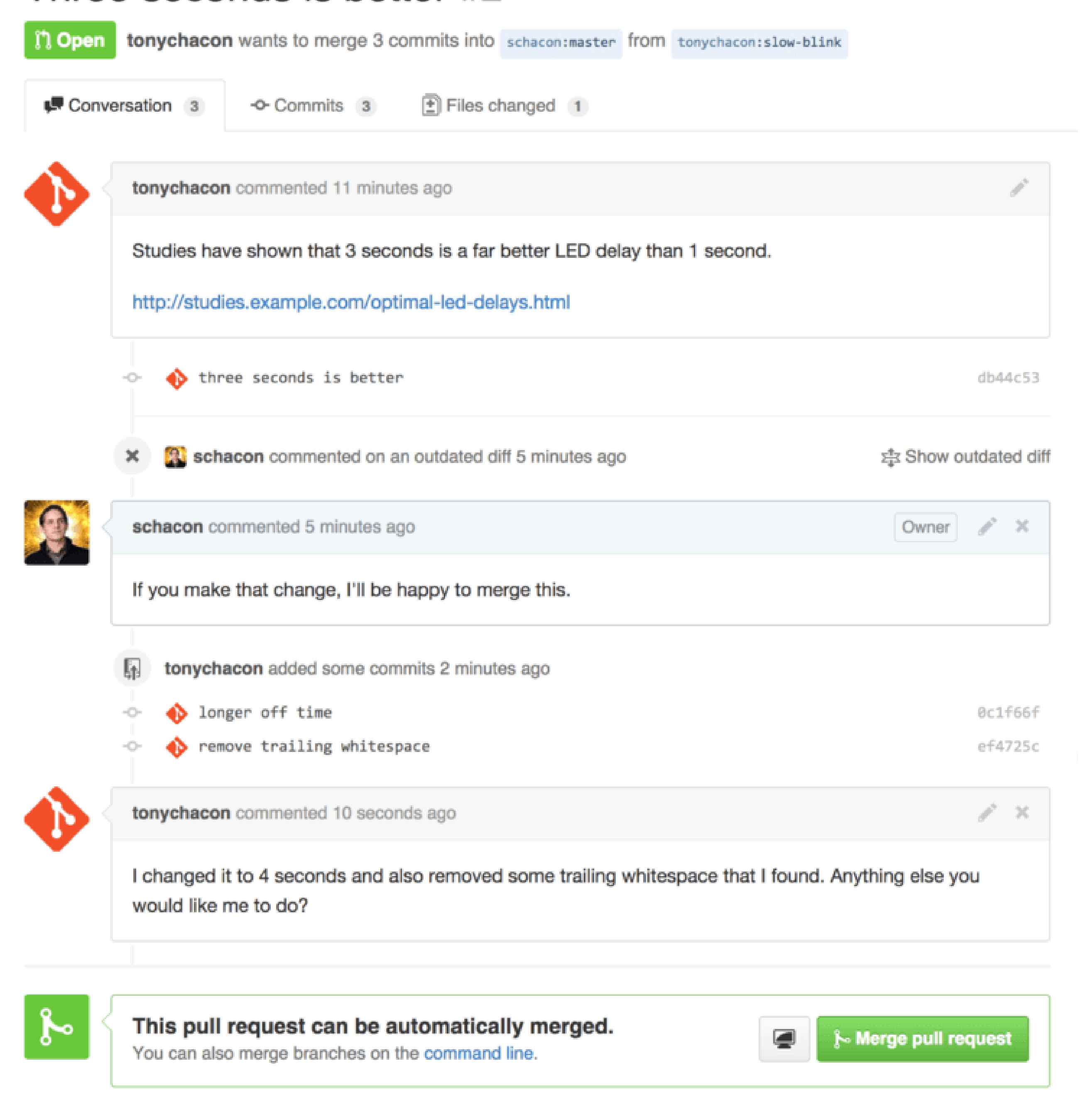


Figure 96. Pull Request final

Es interesante notar que si pulsas en “Files Changed” dentro del Pull Request, verás el “diff unificado”, es decir, los cambios que se introducirían en la rama principal si la otra rama fuera fusionada. En términos de git, lo que hace es mostrarte la salida del comando `git diff master...<rama>`. Mira en Decidiendo qué introducir para saber más sobre este tipo de diff.

Otra cosa interesante es que GitHub también comprueba si el Pull Request se fusionaría limpiamente (de forma automática) dando entonces un botón para hacerlo. Este botón solo lo veremos si además somos los propietarios del repositorio. Si pulsas este botón, GitHub fusionará sin avance rápido, es decir, que incluso si la fusión pudiera ser de tipo avance-rápido, de todas formas crearía un commit de fusión.

Si quieres, puedes obtener la rama en tu equipo y hacer la fusión localmente. Si fusionas esta rama en la rama `master` y la subes a GitHub, el Pull Request se cerrará de forma automática.

Este es el flujo de trabajo básico que casi todos los proyectos de GitHub utilizan. Se crean las ramas de trabajo, se crean con ellas los Pull Requests, se genera una discusión, se añade probablemente más trabajo a la rama y finalmente la petición es cerrada (rechazada) o fusionada.

> **NOTE**
>
> *No solo forks*
>
> Observa que también puedes abrir un Pull Request entre dos ramas del mismo repositorio. Si estás trabajando en una característica con alguien y ambos tenéis acceso de escritura al repositorio, puedes subir una rama al mismo y abrir un Pull Request con ella de fusión con `master` para poder formalizar el proceso de revisión de código y discusión. Para esto no se requieren bifurcaciones (forks).

Pull Requests Avanzados

Ahora que sabemos cómo participar de forma básica en un proyecto de GitHub, veamos algunos trucos más acerca de los Pull Requests que ayudarán a usarlos de forma más eficaz.

Pull Requests como parches

Hay que entender que muchos proyectos no tienen la idea de que los Pull Requests sean colas de parches perfectos que se pueden aplicar limpiamente en orden, como sucede con los proyectos basados en listas de correo. Casi todos los proyectos de GitHub consideran las ramas de Pull Requests como conversaciones evolutivas acerca de un cambio propuesto, culminando en un diff unificado que se aplica fusionando.

Esto es importante, ya que normalmente el cambio se sugiere bastante antes de que el código sea suficientemente bueno, lo que lo aleja bastante del modelo basado en parches por lista de correo. Esto facilita una discusión más temprana con los colaboradores, lo que hace que la llegada de la solución correcta sea un esfuerzo de comunidad. Cuando el cambio llega con un Pull Request y los colaboradores o la comunidad sugieren un cambio, normalmente los parches no son directamente alterados, sino que se realiza un nuevo commit en la rama para enviar la diferencia que materializa esas sugerencias, haciendo avanzar la conversación con el contexto del trabajo previo intacto.

Por ejemplo, si miras de nuevo en Pull Request final, verás que el colaborador no reorganiza su commit y envía un nuevo Pull Request. En lugar, lo que hace es añadir nuevos commits y los envía a la misma rama. De este modo, si vuelves a mirar el Pull Request en el futuro, puedes encontrar fácilmente todo el contexto con todas las decisiones tomadas. Al pulsar el botón “Merge”, se crea un commit de fusión que referencia al Pull Request, con lo que es fácil localizar para revisar la conversación original, si es necesario.

Manteniéndonos actualizados

Si el Pull Request se queda anticuado o por cualquier otra razón no puede fusionarse limpiamente, lo normal es corregirlo para que el responsable pueda fusionarlo fácilmente. GitHub comprobará esto y te dirá si cada Pull Request tiene una fusión trivial posible o no.





Figure 97. Pull Request que no puede fusionarse limpiamente

Si ves algo parecido a Pull Request que no puede fusionarse limpiamente, seguramente prefieras corregir la rama de forma que se vuelva verde de nuevo y el responsable no tenga trabajo extra con ella.

Tienes dos opciones para hacer esto. Puedes, por un lado, reorganizar (rebase) la rama con el contenido de la rama `master` (normalmente esta es la rama desde donde se hizo la bifurcación), o bien puedes fusionar la rama objetivo en la tuya.

Muchos desarrolladores eligen la segunda opción, por las mismas razones que que dijimos en la sección anterior. Lo que importa aquí es la historia y la fusión final, por lo que reorganizar no es mucho más que tener una historia más limpia y, sin embargo, es de lejos más complicado de hacer y con más posibilidad de error.

Si quieres fusionar en la rama objetivo para hacer que tu Pull Request sea fusionable, deberías añadir el repositorio original como un nuevo remoto, bajártelo (fetch), fusionar la rama principal en la tuya, corregir los problemas que surjan y finalmente enviarla (push) a la misma rama donde hiciste la solicitud de integración.

Por ejemplo, supongamos que en el ejemplo “tonychacon” que hemos venido usando, el autor original hace un cambio que crea un conflicto con el Pull Request. Seguiremos entonces los siguientes pasos.

```
$ git remote add upstream https://github.com/schacon/blink ①

$ git fetch upstream ②
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
Unpacking objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
From https://github.com/schacon/blink
 * [new branch]      master     -> upstream/master

$ git merge upstream/master ③
Auto-merging blink.ino
CONFLICT (content): Merge conflict in blink.ino
Automatic merge failed; fix conflicts and then commit the result.

$ vim blink.ino ④
$ git add blink.ino
$ git commit
[slow-blink 3c8d735] Merge remote-tracking branch 'upstream/master' \
    into slower-blink

$ git push origin slow-blink ⑤
Counting objects: 6, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 682 bytes | 0 bytes/s, done.
Total 6 (delta 2), reused 0 (delta 0)
To https://github.com/tonychacon/blink
   ef4725c..3c8d735  slower-blink -> slow-blink
```

① Añadir el repositorio original como un remoto llamado “upstream”

② Obtener del remoto lo último enviado al repositorio

③ Fusionar la rama principal en la nuestra

④ Corregir el conflicto surgido

⑤ Enviar de nuevo los cambios a la rama del Pull Request

Cuando haces esto, el Pull Request se actualiza automáticamente y se re-chequea para ver si es posible un fusionado automático o no.

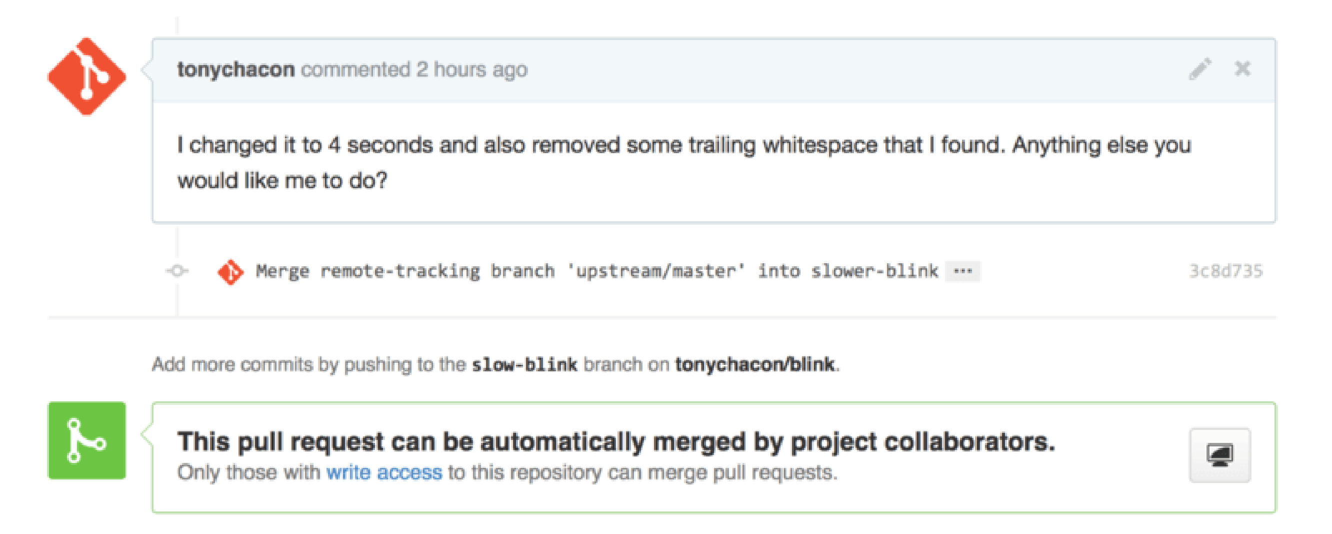


Figure 98. Ahora el Pull Request ya fusiona bien

Una de las cosas interesantes de Git es que puedes hacer esto continuamente. Si tienes un proyecto con mucha historia, puedes fácilmente fusionarte la rama objetivo (`master`) una y otra vez cada vez que sea necesario, evitando conflictos y haciendo que el proceso de integración de tus cambios sea muy manejable.

Si finalmente prefieres reorganizar la rama para limpiarla, también puedes hacerlo, pero se recomienda no forzar el push sobre la rama del Pull Request. Si otras personas se la han bajado y hacen más trabajo en ella, provocarás los problemas vistos en Los Peligros de Reorganizar. En su lugar, envía la rama reorganizada a una nueva rama de GitHub y abre con ella un nuevo Pull Request, con referencia al antiguo, cerrando además éste.

Referencias

La siguiente pregunta puede ser "¿cómo hago una referencia a un Pull Request antiguo?". La respuesta es, de varias formas.

Comencemos con cómo referenciar otro Pull Request o una incidencia (Issue). Todas las incidencias y Pull Requests tienen un número único que los identifica. Este número no se repite dentro de un mismo proyecto. Por ejemplo, dentro de un proyecto solo podemos tener un Pull Request con el número 3, y una incidencia con el número 3. Si quieres hacer referencia al mismo, basta con poner el símbolo `#` delante del número, en cualquier comentario o descripción del Pull Request o incidencia. También se puede poner referencia tipo `usuario#numero` para referirnos a un Pull Request o incidencia en una bifurcación que haya creado ese usuario, o incluso puede usarse la forma `usuario/repo#num` para referirse a una incidencia o Pull Request en otro repositorio diferente.

Veamos un ejemplo. Supongamos que hemos reorganizado la rama del ejemplo anterior, creado un nuevo pull request para ella y ahora queremos hacer una referencia al viejo Pull Request desde el nuevo. También queremos hacer referencia a una incidencia en la bifurcación del repositorio, y una incidencia de un proyecto totalmente distinto. Podemos rellenar la descripción justo como vemos en Referencias cruzadas en un Pull Request..

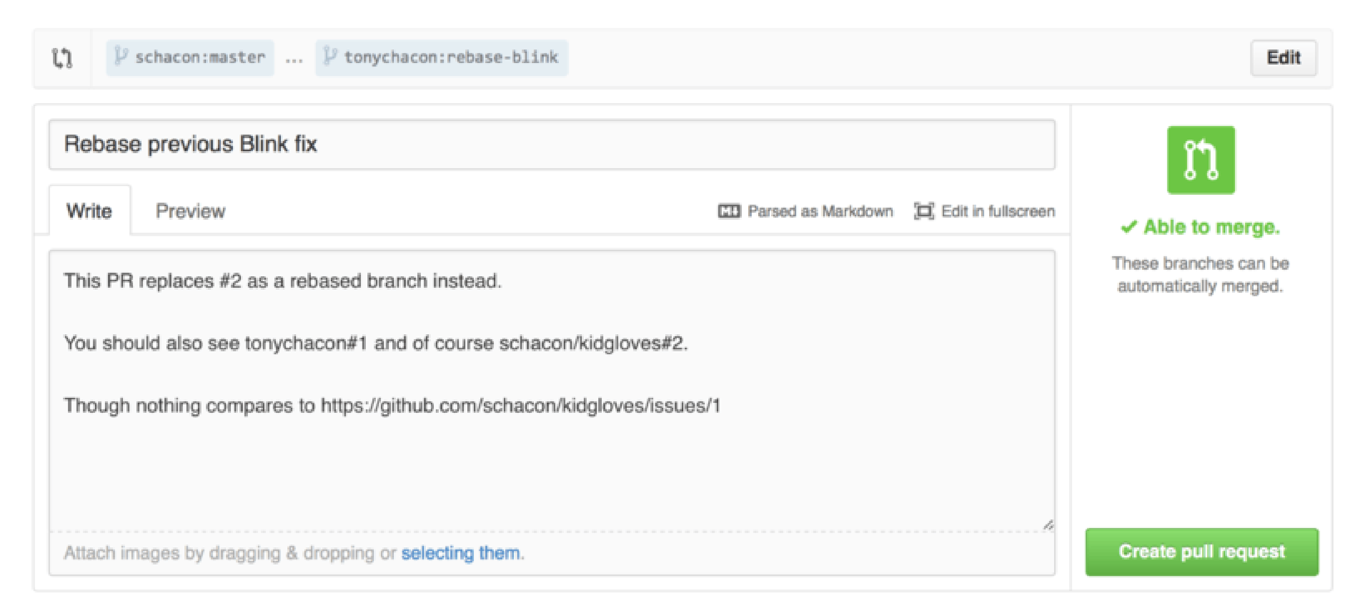


Figure 99. Referencias cruzadas en un Pull Request.

Cuando enviamos este pull request, veremos todo como en Cómo se ven las referencias cruzadas en el Pull Request..

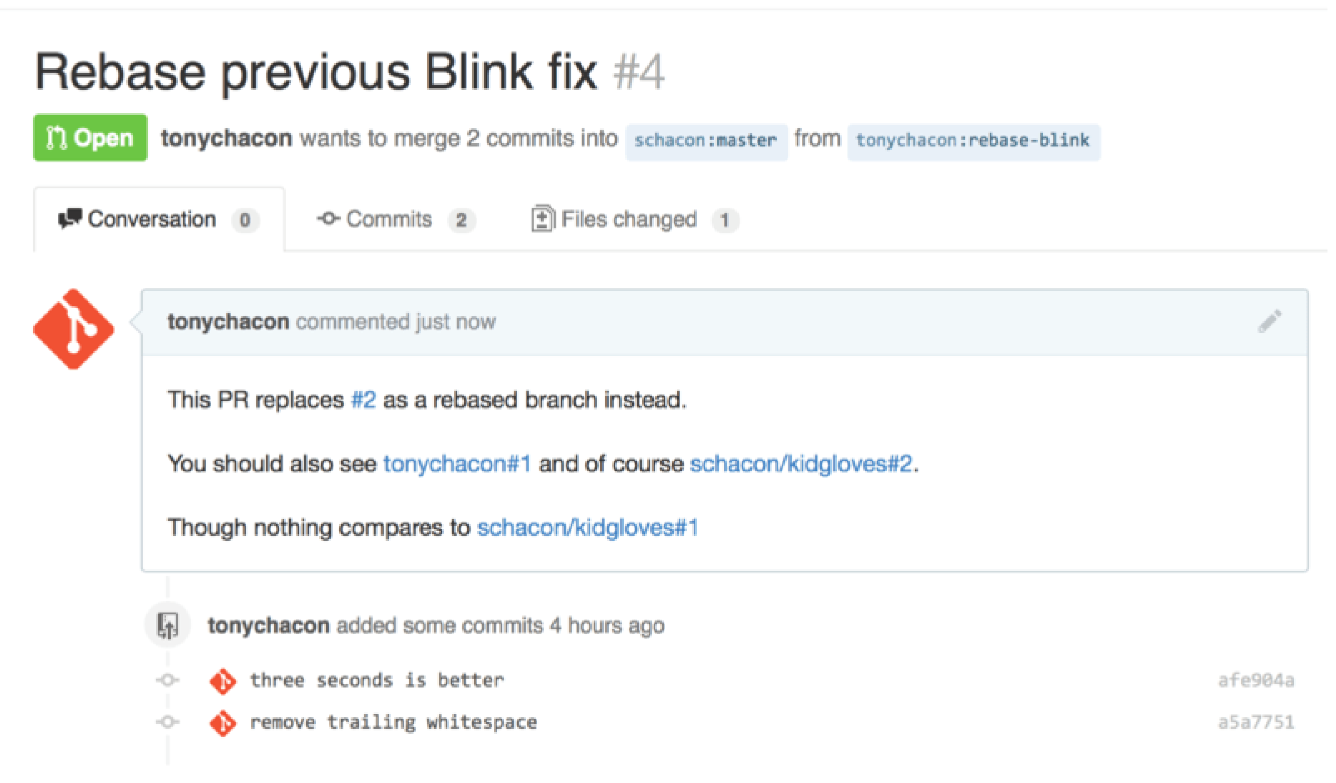


Figure 100. Cómo se ven las referencias cruzadas en el Pull Request.

Observa que la URL completa de GitHub que hemos puesto ahí ha sido acortada a la información que necesitamos realmente.

Ahora, si Tony regresa y cierra el Pull Request original, veremos que GitHub crea un evento en la línea de tiempo del Pull Request. Esto significa que cualquier que visite este Pull Request y vea que está cerrado, puede fácilmente enlazarlo al que lo hizo obsoleto. El enlace se mostrará tal como en Cómo se ven las referencias cruzadas en el Pull Request..

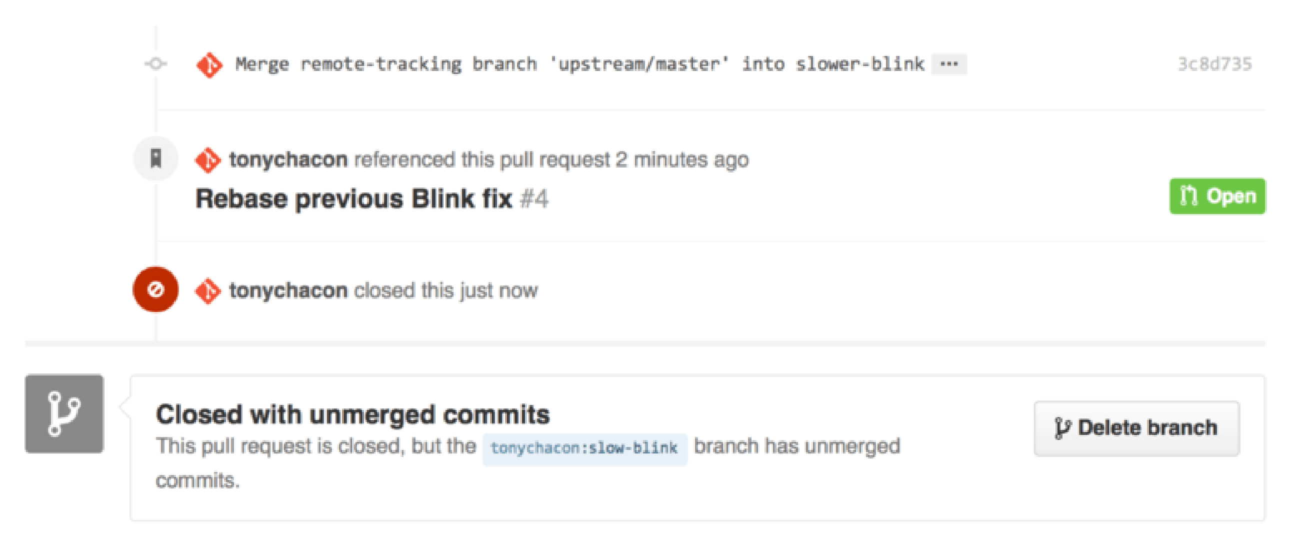


Figure 101. Cómo se ven las referencias cruzadas en el Pull Request.

Además de los números de incidencia, también puedes hacer referencia a un commit específico usando la firma SHA-1. Puedes utilizar la cadena SHA-1 completa (de 40 caracteres) y al detectarla GitHub en un comentario, la convertirá automáticamente en un enlace directo al commit. Nuevamente, puedes hacer referencia a commits en bifurcaciones o en otros repositorios del mismo modo que hicimos con las incidencias.

Markdown

En enlazado a otras incidencias es solo el comienzo de las cosas interesantes que se pueden hacer con cualquier cuadro de texto de GitHub. En las descripciones de las incidencias y los Pull Requests, así como en los comentarios, y otros cuadros de texto, se puede usar lo que se conoce "formato Markdown de GitHub". El formato Markdown es como escribir en texto plano pero que luego se convierte en texto con formato.

Mira en Ejemplo de texto en Markdown y cómo queda después. un ejemplo de cómo los comentarios o el texto puede escribirse y luego formatearse con Markdown.

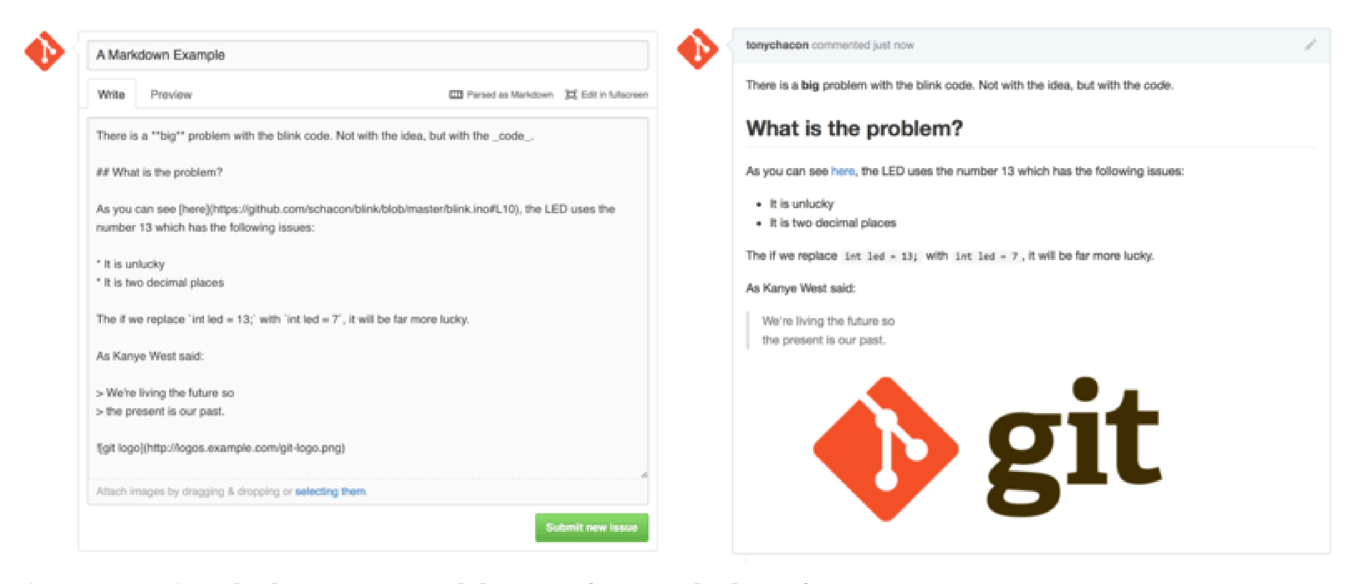


Figure 102. Ejemplo de texto en Markdown y cómo queda después.

El formato Markdown de GitHub

En GitHub se añaden algunas cosas a la sintaxis básica del Markdown. Son útiles al tener relación con los Pull Requests o las incidencias.

Listas de tareas

La primera característica añadida, especialmente interesante para los Pull Requests, son las listas de tareas. Una lista de tareas es una lista de cosas con su marcador para indicar que han terminado. En un Pull Requests o una incidencia nos sirven para anotar la lista de cosas pendientes para considerar terminado el trabajo relacionado con esa incidencia.

Puedes crear una lista de tareas así:

```
- [X] Write the code
- [ ] Write all the tests
- [ ] Document the code
```

Si incluimos esto en la descripción de nuestra incidencia o Pull Request, lo veremos con el aspecto de Cómo se ven las listas de tareas de Markdown.

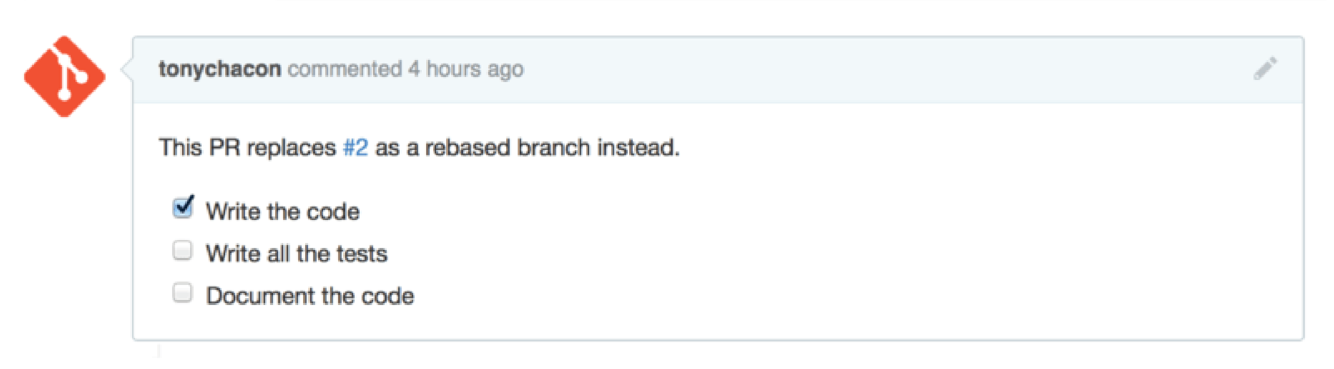


Figure 103. Cómo se ven las listas de tareas de Markdown.

Esto se suele usar en Pull Requests para indicar qué cosas hay que hacer en la rama antes de considerar que el Pull Request está listo para fusionarse. La parte realmente interesante es que puedes pulsar los marcadores para actualizar el comentario indicando qué tareas se finalizaron, sin necesidad de editar el texto markdown del mismo.

Además, GitHub mostrará esas listas de tareas como metadatos de las páginas que las muestran. Por ejemplo, si tienes un Pull Request con tareas y miras la página resumen de todos los Pull Request, podrás ver cuánto trabajo queda pendiente. Esto ayuda a la gente a dividir los Pull Requests en subtareas y ayuda a otras personas a seguir la evolución de la rama. Se puede ver un ejemplo de esto en Resumen de lista de tareas en la lista de PR..



Figure 104. Resumen de lista de tareas en la lista de PR.

Esto es increíblemente útil cuando se abre un Pull Request al principio y se quiere usar para seguir el progreso de desarrollo de la característica.

Fragmentos de código

También se pueden añadir fragmentos de código a los comentarios. Esto resulta útil para mostrar algo que te gustaría probar antes de ponerlo en un commit de tu rama. Esto también se suele usar para añadir ejemplos de código que no funciona u otros asuntos.

Para añadir un fragmento de código, lo tienes que encerrar entre los símbolos del siguiente ejemplo.

````
```java
for(int i=0 ; i < 5 ; i++)
{
   System.out.println("i is : " + i);
}
```
````

Si añades junto a los símbolos el nombre de un lenguaje de programación, como hacemos aquí con *java*, GitHub intentará hacer el resaltado de la sintaxis del lenguaje en el fragmento. En el caso anterior, quedaría con el aspecto de Cómo se ve el fragmento de código..



Figure 105. Cómo se ve el fragmento de código.

Citas

Si estás respondiendo a un comentario grande, pero solo a una pequeña parte, puedes seleccionar la parte que te interesa y citarlo, para lo que precedes cada línea citada del símbolo `>`. Esto es tan útil que hay un atajo de teclado para hacerlo: si seleccionas el texto al que quieres contestar y pulsas la tecla `r`, creará una cita con ese texto en la caja del comentario.

Un ejemplo de cita:

```
> Whether 'tis Nobler in the mind to suffer
> The Slings and Arrows of outrageous Fortune,

How big are these slings and in particular, these arrows?
```

Una vez introducida, el comentario se vería como en Rendered quoting example..

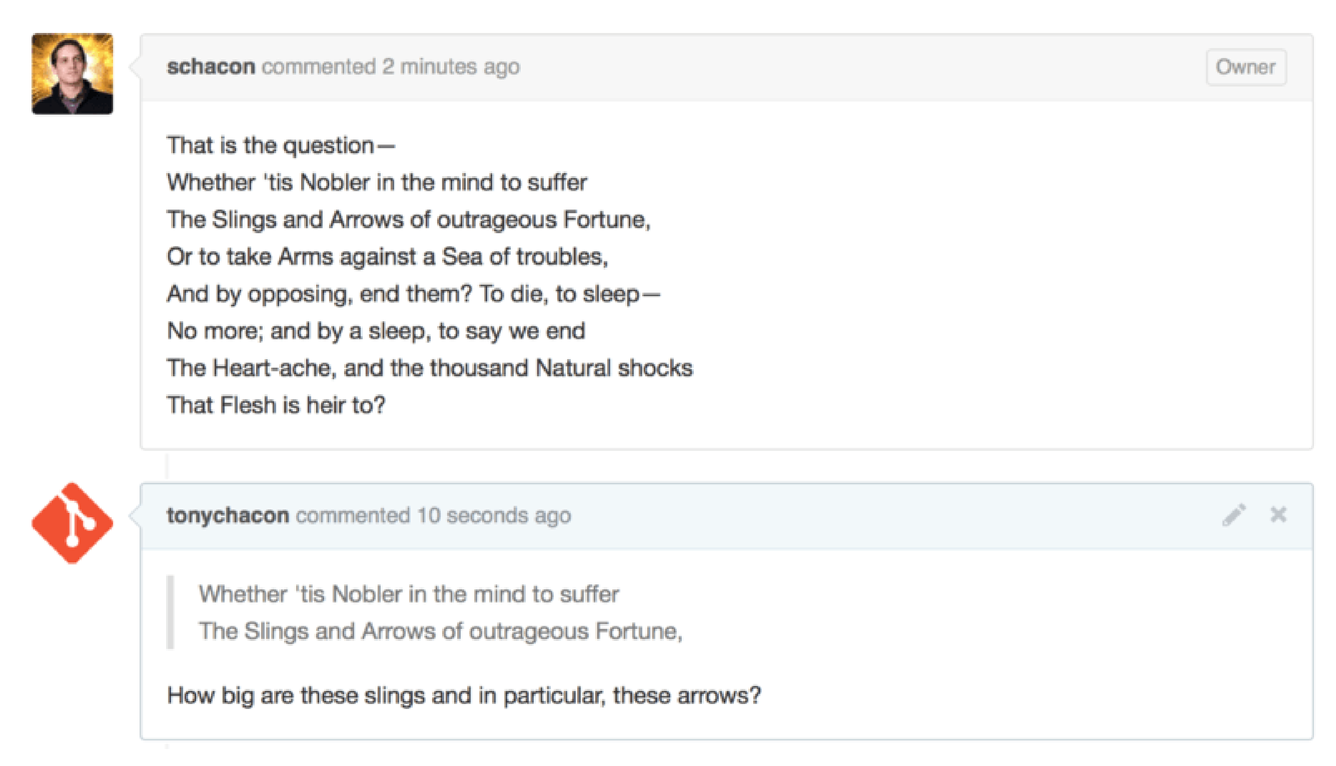


Figure 106. Rendered quoting example.

Emojis (emoticonos)

Finalmente, también puedes usar emoji (emoticonos) en tus comentarios. Se utiliza mucho en las discusiones de las incidencias y Pull Requests de GitHub. Incluso tenemos un asistente de emoji: si escribes un comentario y tecleas el carácter `:`, verás cómo aparecen iconos para ayudarte a completar el que quieras poner.

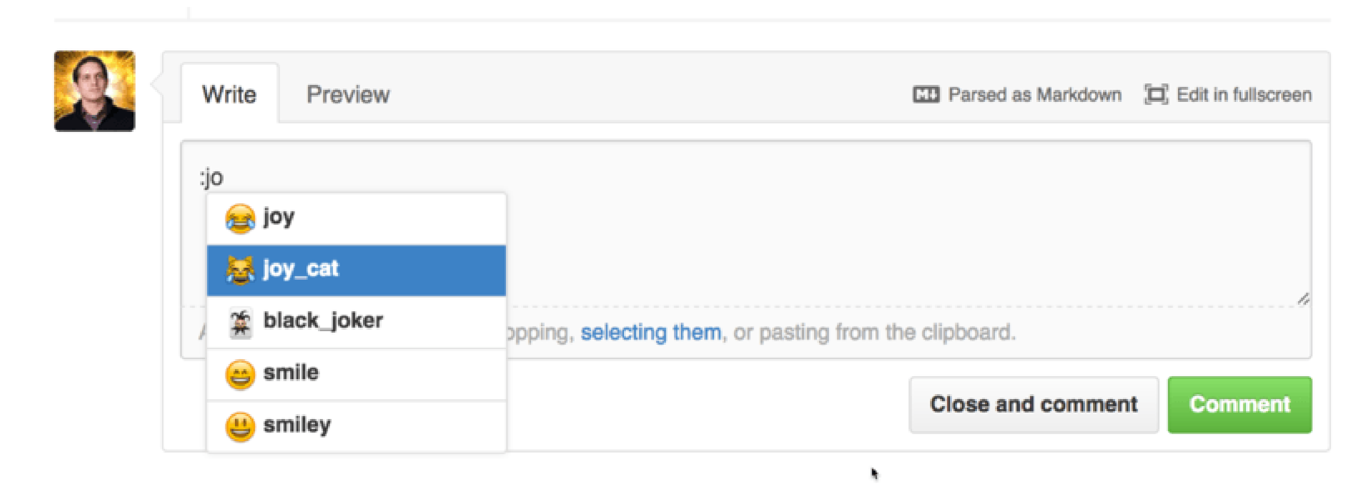


Figure 107. Emoji auto-completando emoji.

Los emoticonos son de la forma `:nombre:` en cualquier punto del comentario. Por ejemplo, podráis escribir algo como esto:

```
I :eyes: that :bug: and I :cold_sweat:.

:trophy: for :microscope: it.

:+1: and :sparkles: on this :ship:, it's :fire::poop:!

:clap::tada::panda_face:
```

Al introducir el comentario, se mostraría como Comentando con muchos emoji..

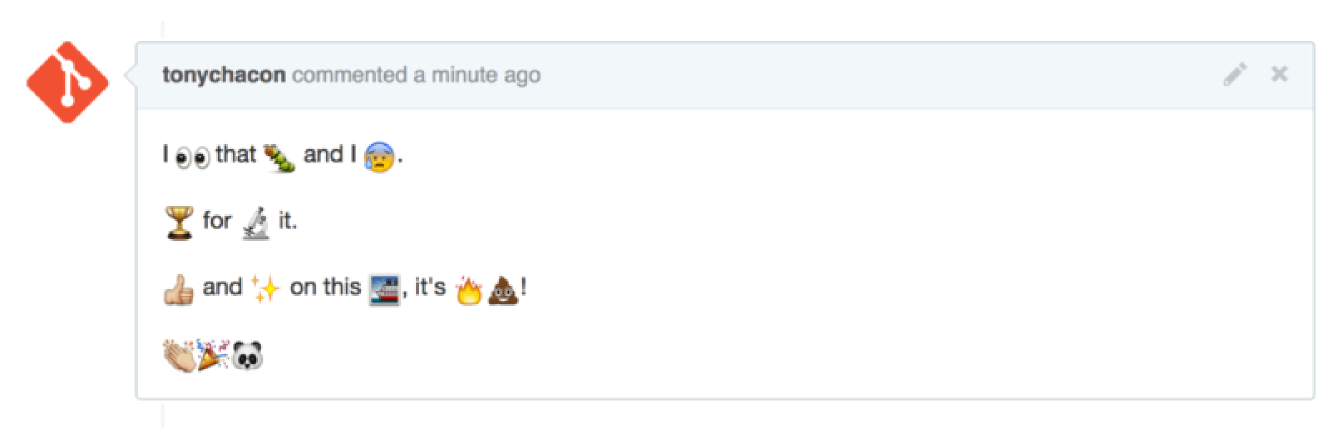


Figure 108. Comentando con muchos emoji.

No es que sean especialmente útiles, pero añaden un elemento de gracia y emoción a un medio en el que de otro modo sería mucho más complicado transmitir las emociones.

NOTE Actualmente hay bastantes sitios web que usan los emoticonos. Hay una referencia interesante para encontrar el emoji que necesitas en cada momento:

http://www.emoji-cheat-sheet.com

Imágenes

Esto no es técnicamente parte de las mejoras a Markdown de GitHub, pero es increíblemente útil. En adición a añadir enlaces con imágenes en el formato Markdown a los comentarios, GitHub permite arrastrar y soltar imágenes en las áreas de texto para insertarlas.

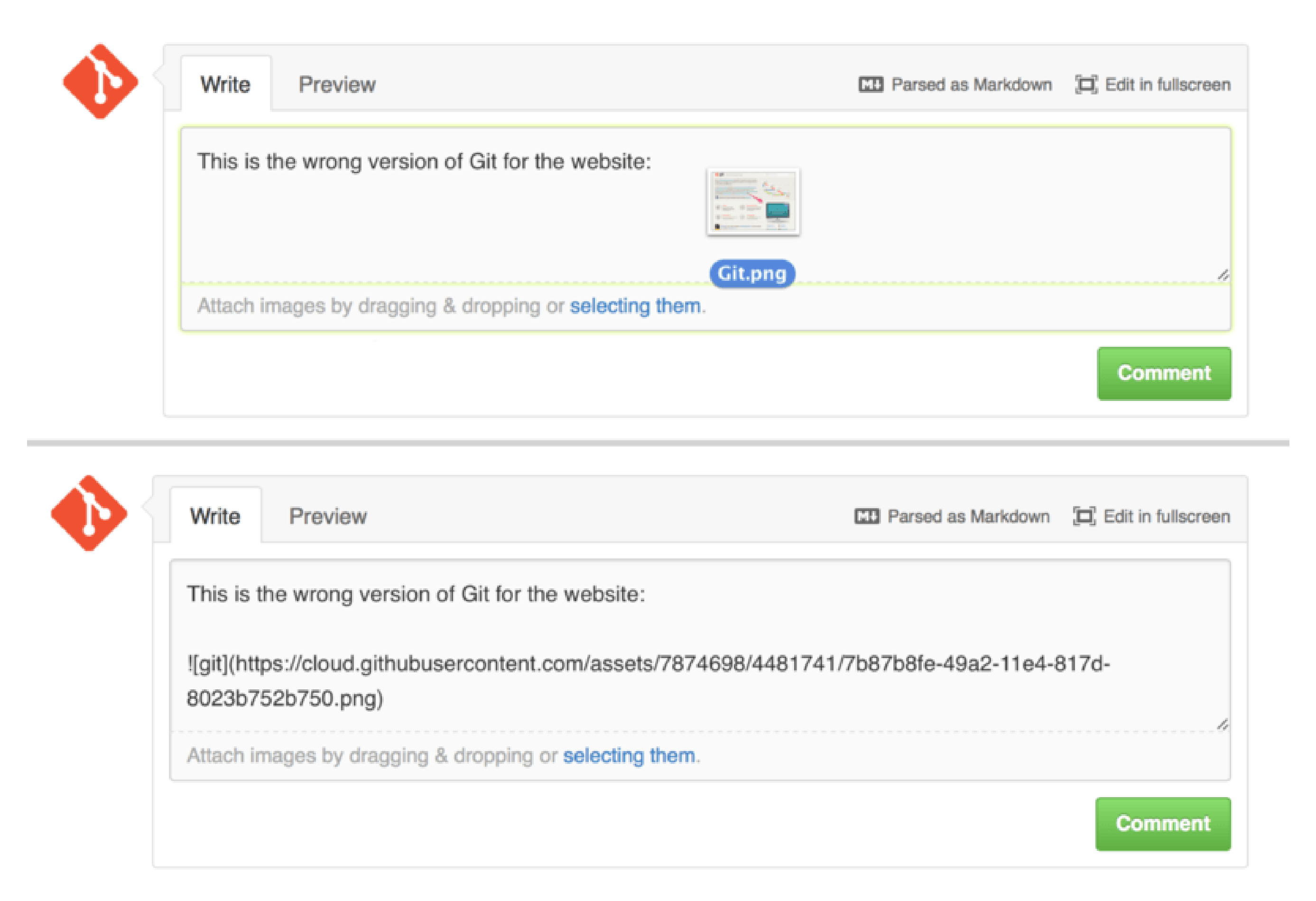


Figure 109. Arrastrar y soltar imágenes para subirlas.

Si vuelves a Referencias cruzadas en un Pull Request., verás una pequeña nota sobre el área de texto “Parsed as Markdown”. Si pulsas ahí te dará una lista completa de cosas que puedes hacer con el formato Markdown de GitHub.

Mantenimiento de un proyecto

Ahora que ya sabes cómo ayudar a un proyecto, veamos el otro lado: cómo puedes crear, administrar y mantener tu propio proyecto.

Creación de un repositorio

Vamos a crear un nuevo repositorio para compartir nuestro código en él. Comienza pulsando el botón “New repository” en el lado derecho de tu página principal, o bien desde el botón + en la barra de botones cercano a tu nombre de usuario, tal como se ve en Desplegable “New repository”..

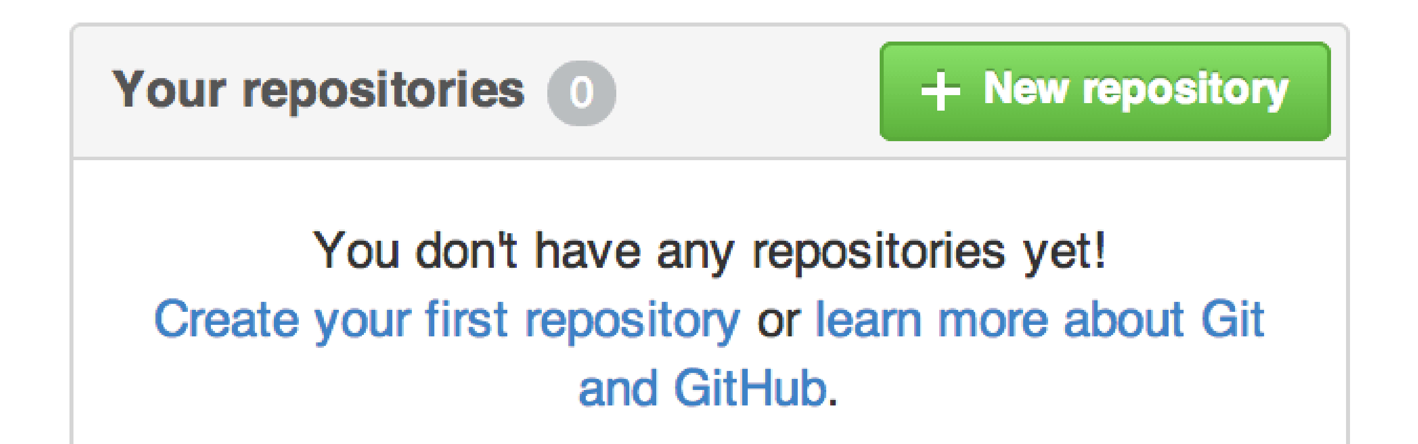


Figure 110. La zona “Your repositories”.

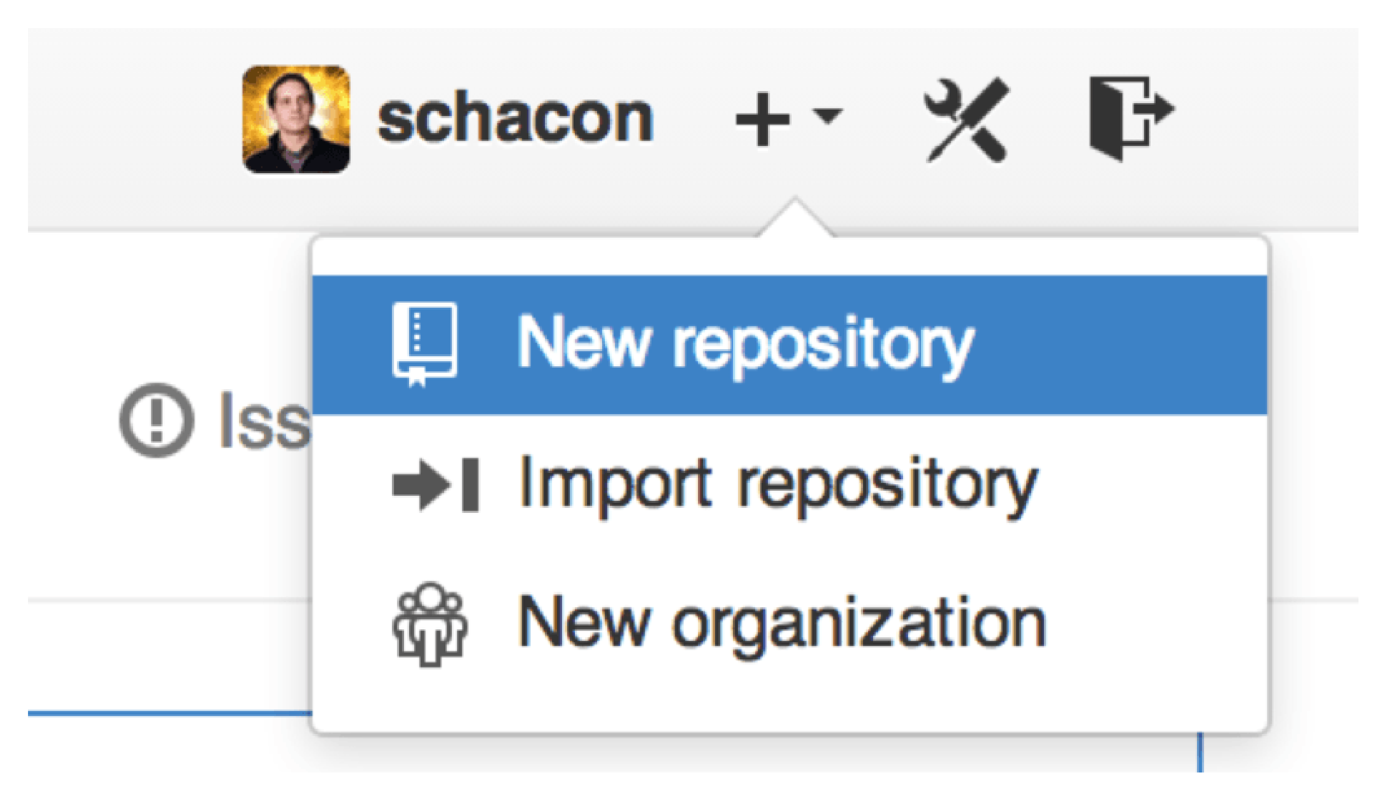


Figure 111. Desplegable “New repository”.

Esto te llevará al formulario para crear un nuevo repositorio:

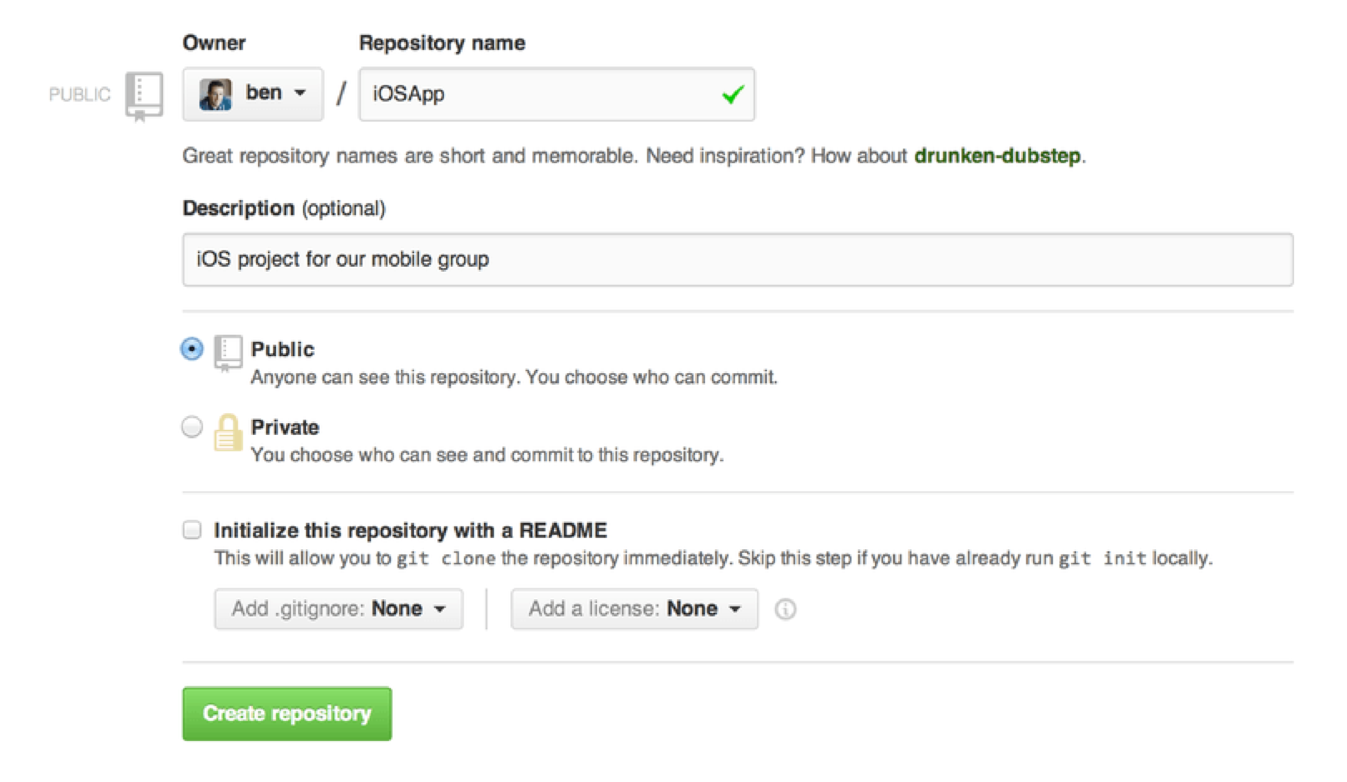


Figure 112. Formulario para crear repositorio.

Todo lo que tienes que hacer aquí es darle un nombre al proyecto; el resto de campos es totalmente opcional. Por ahora, pulsa en el botón “Create Repository” y listo: se habrá creado el repositorio en GitHub, con el nombre `<usuario>/<proyecto>`

Dado que no tiene todavía contenido, GitHub te mostrará instrucciones para crear el repositorio Git, o para conectarlo a un proyecto Git existente. No entraremos aquí en esto; si necesitas refrescarlo, revisa el capítulo Fundamentos de Git.

Ahora que el proyecto está alojado en GitHub, puedes dar la URL a cualquiera con quien quieras compartirlo. Cada proyecto en GitHub es accesible mediante HTTP como `https://github.com/<usuario>/<proyecto>`, y también con SSH con la dirección `git@github.com:<usuario>/<proyecto>`. Git puede obtener y enviar cambios en ambas URL, ya que tienen control de acceso basado en las credenciales del usuario.

NOTE	Suele ser preferible compartir la URL de tipo HTTP de los proyectos públicos, puesto que así el usuario no necesitará una cuenta GitHub para clonar el proyecto. Si das la dirección SSH, los usuarios necesitarán una cuenta GitHub y subir una clave SSH para acceder. Además, la URL HTTP es exactamente la misma que usamos para ver la página web del proyecto.

Añadir colaboradores

Si estás trabajando con otras personas y quieres darle acceso de escritura, necesitarás añadirlas como “colaboradores”. Si Ben, Jeff y Louise se crean cuentas en GitHub, y quieres darles acceso de escritura a tu repositorio, los tienes que añadir al proyecto. Al hacerlo le darás permiso de “push”, que significa que tendrán tanto acceso de lectura como de escritura, en el proyecto y en el repositorio Git.

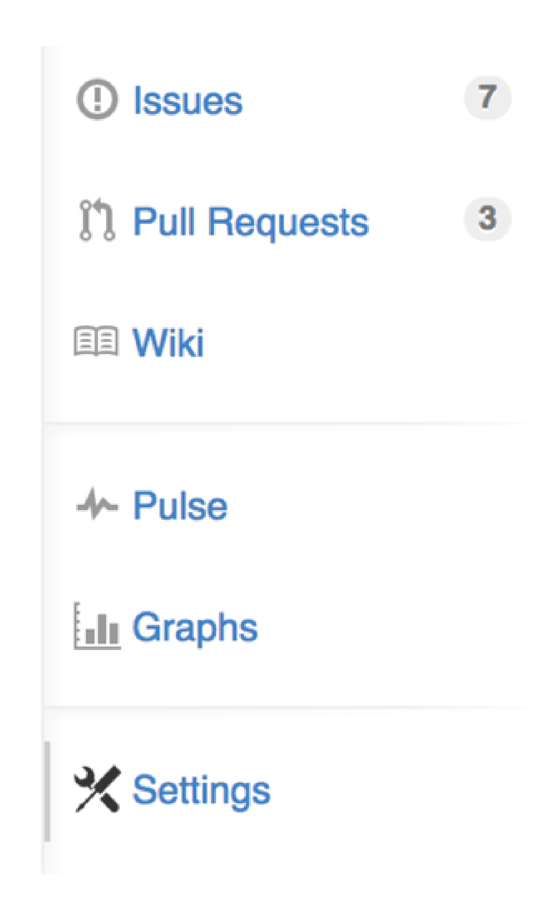


Figure 113. Enlace a ajustes del repositorio.

Selecciona “Collaborators” del menú del lado izquierdo. Simplemente, teclea el usuario en la caja, y pulsa en “Add collaborator.” Puedes repetir esto las veces que necesites para dar acceso a otras personas. Si necesitas quitar un acceso, pulsa en la “X” del lado derecho del usuario.

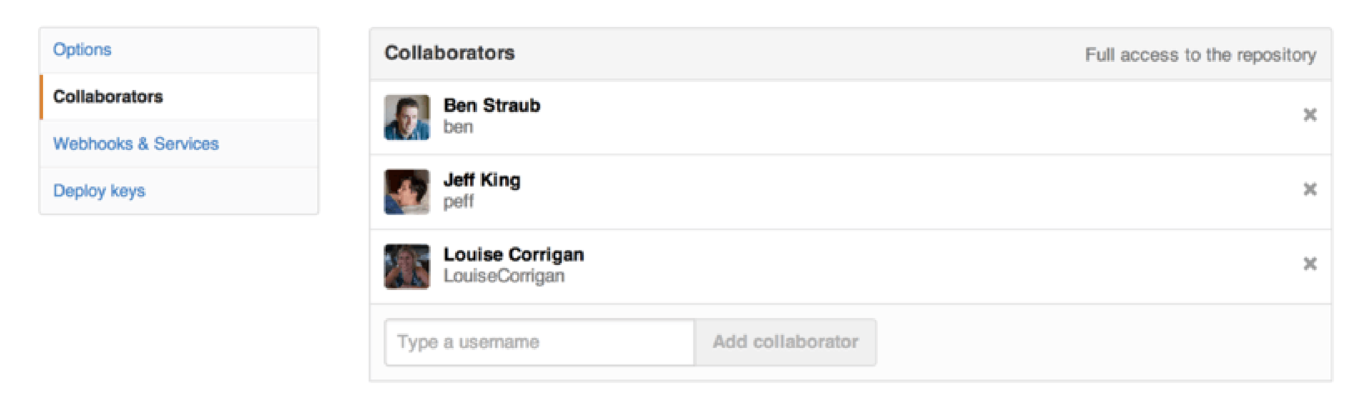


Figure 114. Colaboradores del repositorio.

Gestión de los Pull Requests

Ahora que tienes un proyecto con algo de código, y probablemente algunos colaboradores con acceso de escritura, veamos qué pasa cuando alguien te hace un Pull Request.

Los Pull Requests pueden venir de una rama en una bifurcación del repositorio, o pueden venir de una rama pero del mismo repositorio. La única diferencia es que, en el primer, caso procede de gente que no tiene acceso de escritura a tu proyecto y quiere integrar en el tuyo cambios interesantes, mientras que en el segundo caso procede de gente con acceso al repositorio.

En los siguientes ejemplos, supondremos que eres “tonychacon” y has creado un nuevo proyecto

para Arduino llamado “fade”.

Notificaciones por correo electrónico

Cuando alguien realiza un cambio en el código y te crea un Pull Request, debes recibir una notificación por correo electrónico avisándote, con un aspecto similar a Notificación por correo de nuevo Pull Request..

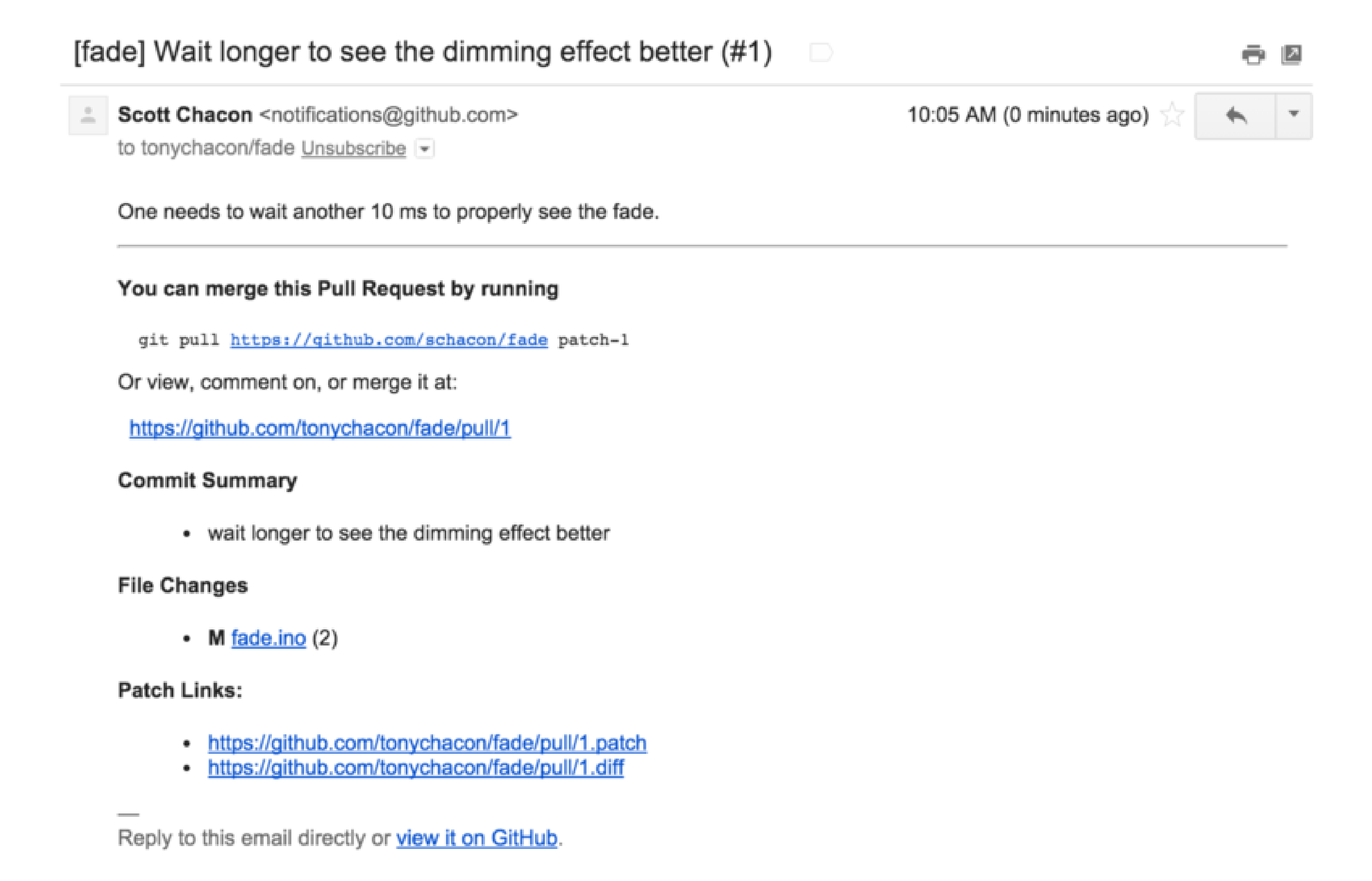


Figure 115. Notificación por correo de nuevo Pull Request.

Hay algunas cosas a destacar en este correo. En primer lugar, te dará un pequeño diffstat (es decir, una lista de archivos cambiados y en qué medida). Además, trae un enlace al Pull Request y algunas URL que puedes usar desde la línea de comandos.

Si observas la línea que dice `git pull <url> patch-1`, es una forma simple de fusionar una rama remota sin tener que añadirla localmente. Lo vimos esto rápidamente en Recuperando ramas remotas. Si lo deseas, puedes crear y cambiar a una rama y luego ejecutar el comando para fusionar los cambios del Pull Request.

Las otras URL interesantes son las de `.diff` y `.patch`, que como su nombre indican, proporcionan diff unificados y formatos de parche del Pull Request. Técnicamente, podrías fusionar con algo como:

```
$ curl http://github.com/tonychacon/fade/pull/1.patch | git am
```

Colaboración en el Pull Request

Como hemos visto en El Flujo de Trabajo en GitHub, puedes participar en una discusión con la persona que generó el Pull Request. Puedes comentar líneas concretas de código, comentar commits completos o comentar el Pull Request en sí mismo, utilizando donde quieras el formato Markdown.

Cada vez que alguien comenta, recibirás nuevas notificaciones por correo, lo que te permite vigilar todo lo que pasa. Cada correo tendrá un enlace a la actividad que ha tenido lugar, y además puedes responder al comentario simplemente contestando al correo.

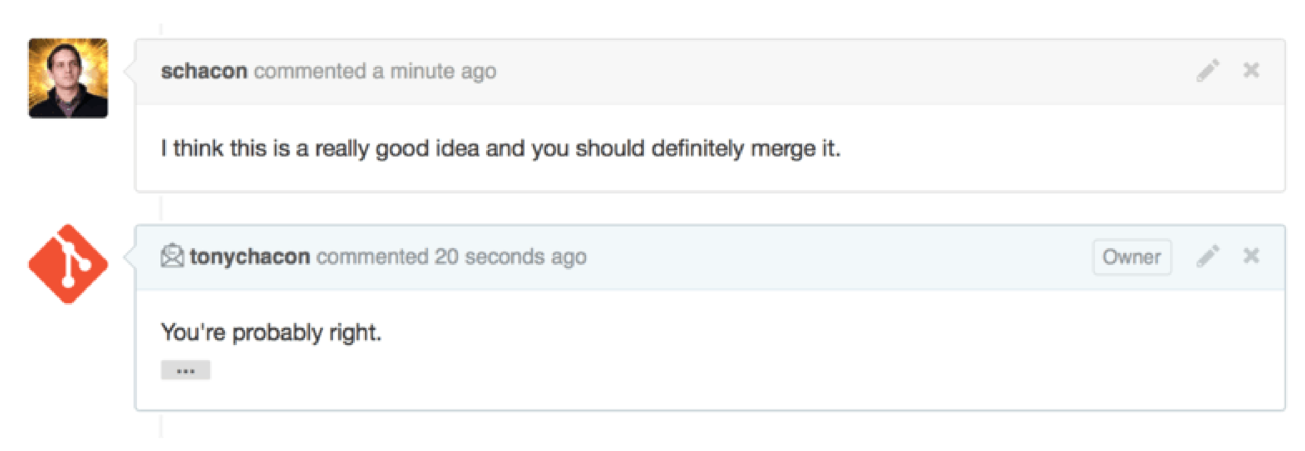


Figure 116. Las respuestas a correos se incluyen en el hilo de discusión.

Una vez que el código está como quieres y quieres fusionarlo, puedes copiar el código y fusionarlo localmente, mediante la sintaxis ya conocida de `git pull <url> <branch>`, o bien añadiendo el fork como nuevo remoto, bajándotelo y luego fusionándolo.

Si la fusión es trivial, también puedes pulsar el botón “Merge” en GitHub. Esto realizará una fusión “sin avance rápido”, creando un commit de fusión incluso si era posible una fusión con avance rápido. Esto significa que cada vez que pulses el botón Merge, se creará un commit de fusión. Como verás en Botón Merge e instrucciones para fusionar manualmente un Pull Request., GitHub te da toda esta información si pulsas el el enlace de ayuda.

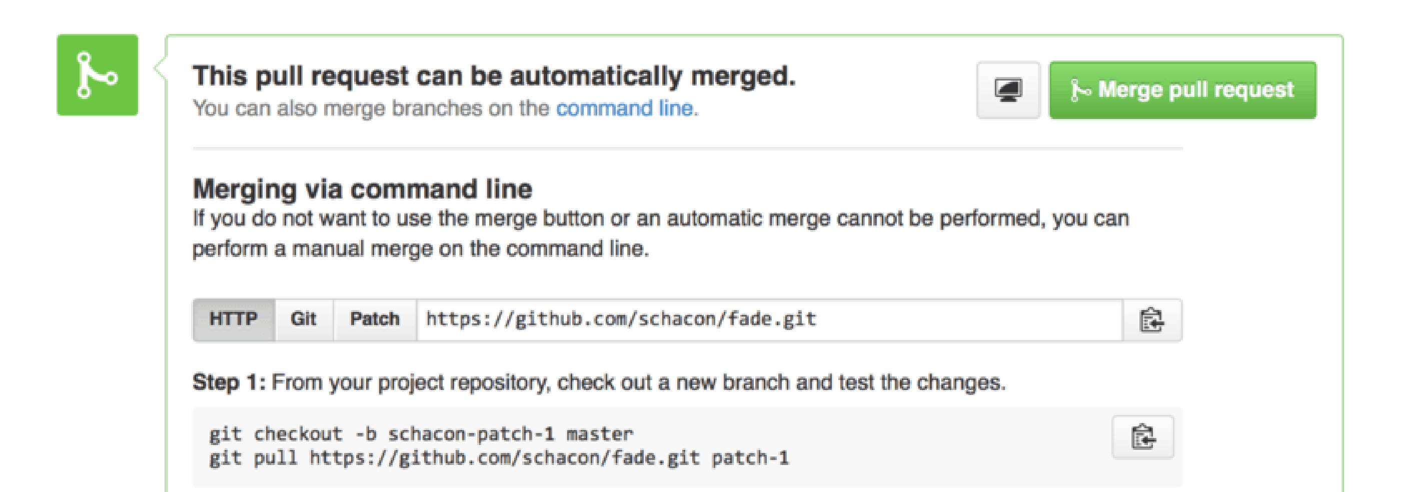


Figure 117. Botón Merge e instrucciones para fusionar manualmente un Pull Request.

Si decides que no quieres fusionar, también puedes cerrar el Pull Request y la persona que lo creó será notificada.

Referencias de Pull Request

Si tienes muchos Pull Request y no quieres añadir un montón de remotos o hacer muchos cada vez, hay un pequeño truco que GitHub te permite. Es un poco avanzado y lo veremos en detalle después en Las especificaciones para hacer referencia a… (refspec), pero puede ser bastante útil.

En GitHub tenemos que las ramas de Pull Request son una especie de pseudo-ramas del servidor. De forma predeterminada no las obtendrás cuando hagas un clonado, pero hay una forma algo oscura de acceder a ellos.

Para demostrarlo, usaremos un comando de bajo nivel (conocido como de “fontanería”, sabremos más sobre esto en Fontanería y porcelana) llamado `ls-remote`. Este comando no se suele usar en el día a día de Git pero es útil para ver las referencias presentes en el servidor.

Si ejecutamos este comando sobre el repositorio “blink” que hemos estado usando antes, obtendremos una lista de ramas, etiquetas y otras referencias del repositorio.

```
$ git ls-remote https://github.com/schacon/blink
10d539600d86723087810ec636870a504f4fee4d    HEAD
10d539600d86723087810ec636870a504f4fee4d    refs/heads/master
6a83107c62950be9453aac297bb0193fd743cd6e    refs/pull/1/head
afe83c2d1a70674c9505cc1d8b7d380d5e076ed3    refs/pull/1/merge
3c8d735ee16296c242be7a9742ebfbc2665adec1    refs/pull/2/head
15c9f4f80973a2758462ab2066b6ad9fe8dcf03d    refs/pull/2/merge
a5a7751a33b7e86c5e9bb07b26001bb17d775d1a    refs/pull/4/head
31a45fc257e8433c8d8804e3e848cf61c9d3166c    refs/pull/4/merge
```

Por supuesto, si estás en tu repositorio y tecleas `git ls-remote origin` podrás ver algo similar pero para el remoto etiquetado como `origin`.

Si el repositorio está en GitHub y tienes Pull Requests abiertos, tendrás estas referencias con el prefijo `refs/pull`. Básicamente, son ramas, pero ya que no están bajo `refs/heads/`, no las obtendrás normalmente cuando clonas o te bajas el repositorio del servidor, ya que el proceso de obtención las ignora.

Hay dos referencias por cada Pull Request, la que termina en `/head` apunta exactamente al último commit de la rama del Pull Request. Así si alguien abre un Pull Request en el repositorio y su rama se llama `bug-fix` apuntando al commit `a5a775`, en nuestro repositorio no tendremos una rama `bug-fix` (puesto que está en el fork) pero tendremos el `pull/<pr#>/head` apuntando a `a5a775`. Esto significa que podemos obtener fácilmente cada Pull Request sin tener que añadir un montón de remotos.

Ahora puedes obtenerlo directamente.

```
$ git fetch origin refs/pull/958/head
From https://github.com/libgit2/libgit2
 * branch            refs/pull/958/head -> FETCH_HEAD
```

Esto dice a Git, “Conecta al remoto `origin` y descarga la referencia llamada `refs/pull/958/head`.” Git obedece y descarga todo lo necesario para construir esa referencia, y deja un puntero al commit que quieres bajo `.git/FETCH_HEAD`. Puedes realizar operaciones como `git merge FETCH_HEAD` aunque el mensaje del commit será un poco confuso. Además, si estás revisando un montón de pull requests, se convertirá en algo tedioso.

Hay también una forma de obtener *todos* los pull requests, y mantenerlos actualizados cada vez que conectas al remoto. Para ello abre el archivo `.git/config` y busca la línea `origin`. Será similar a esto:

```
[remote "origin"]
    url = https://github.com/libgit2/libgit2
    fetch = +refs/heads/*:refs/remotes/origin/*
```

La línea que comienza con `fetch =` es un “refspec.” Es una forma de mapear nombres del remoto con nombres de tu copia local. Este caso concreto dice a Git, que "las cosas en el remoto bajo `refs/heads` deben ir en mi repositorio bajo `refs/remotes/origin`." Puedes modificar esta sección añadiendo otra refspec:

```
[remote "origin"]
    url = https://github.com/libgit2/libgit2.git
    fetch = +refs/heads/*:refs/remotes/origin/*
    fetch = +refs/pull/*/head:refs/remotes/origin/pr/*
```

Con esta última línea decimos a Git, “Todas las referencias del tipo `refs/pull/123/head` deben

guardarse localmente como `refs/remotes/origin/pr/123`.” Ahora, si guardas el archivo y ejecutas un `git fetch` tendremos:

```
$ git fetch
# …
 * [new ref]         refs/pull/1/head -> origin/pr/1
 * [new ref]         refs/pull/2/head -> origin/pr/2
 * [new ref]         refs/pull/4/head -> origin/pr/4
# …
```

Ya tienes todos los pull request en local de forma parecida a las ramas; son solo-lectura y se actualizan cada vez que haces un fetch. Pero hace muy fácil probar el código de un pull request en local:

```
$ git checkout pr/2
Checking out files: 100% (3769/3769), done.
Branch pr/2 set up to track remote branch pr/2 from origin.
Switched to a new branch 'pr/2'
```

La referencia `refs/pull/#/merge` de GitHub representa el commit que resultaría si pulsamos el botón “merge”. Esto te permite probar la fusión del pull request sin llegar a pulsar dicho botón.

Pull Requests sobre Pull Requests

No solamente se puede abrir Pull Requests en la rama `master`, también se pueden abrir sobre cualquier rama de la red. De hecho, puedes poner como objetivo otro Pull Request.

Si ves que un Pull Request va en la buena dirección y tienes una idea para hacer un cambio que depende de él, o bien no estás seguro de que sea una buena idea, o no tienes acceso de escritura en la rama objetivo, puedes abrir un Pull Request directamente.

Cuando vas a abrir el Pull Request, hay una caja en la parte superior de la página que especifica qué rama quieres usar y desde qué rama quieres hacer la petición. Si pulsas el botón “Edit” en el lado derecho de la caja, puedes cambiar no solo las ramas sino también la bifurcación.

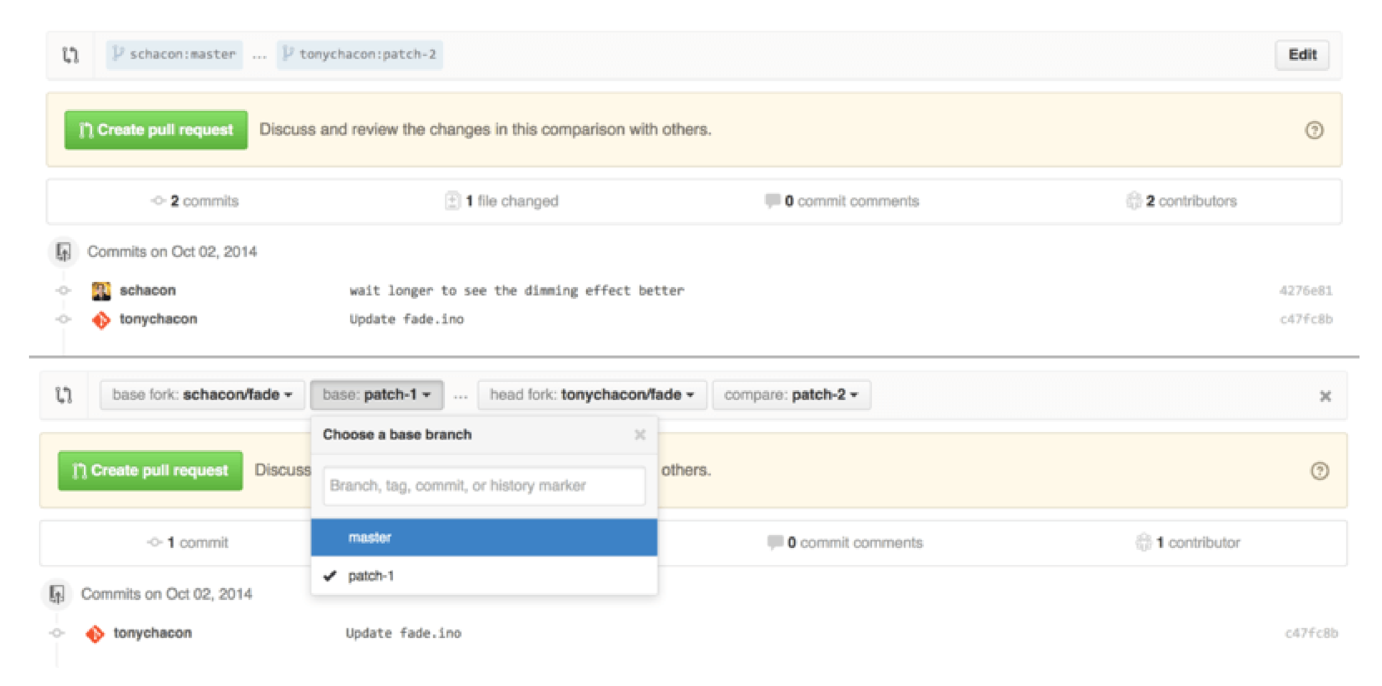


Figure 118. Cambio manual de la rama o del fork en un pull request.

Aquí puedes fácilmente especificar la fusión de tu nueva rama en otro Pull Request o en otrá bifurcación del proyecto.

Menciones y notificaciones

GitHub tiene un sistema de notificaciones que resulta útil cuando necesitas pedir ayuda o necesitas la opinión de otros usuarios o equipos concretos.

En cualquier comentario, si comienzas una palabra comenzando con el carácter `@`, intentará auto-completar nombres de usuario de personas que sean colaboradores o responsables en el proyecto.

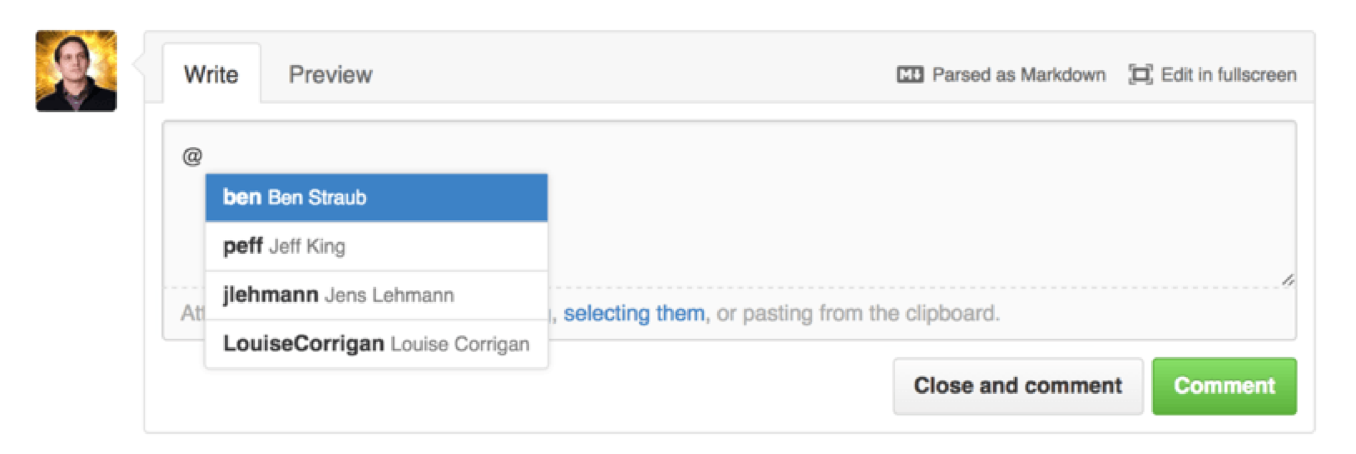


Figure 119. Empieza tecleando @ para mencionar a alguien.

También puedes mencionar a un usuario que no esté en la lista desplegable, pero normalmente el autocompletado lo hará más rápido.

Una vez que envías un comentario con mención a un usuario, el usuario citado recibirá una notificación. Es decir, es una forma de implicar más gente en una conversación. Esto es muy común en los Pull Requests para invitar a terceros a que participen en la revisión de una incidencia o un pull request.

Si alguien es mencionado en un Pull request o incidencia, quedará además “suscrito” y recibirá desde este momento las notificaciones que genere su actividad. Del mismo modo, el usuario que crea la incidencia o el Pull request queda automáticamente suscrito para recibir las notificaciones, disponiendo todos de un botón “Unsubscribe” para dejar de recibirlas.



Figure 120. Quitar suscripción de un pull request o incidencia.

Página de notificaciones

Cuando decimos “notificaciones”, nos referimos a una forma por la que GitHub intenta contactar contigo cuando tienen lugar eventos, y éstas pueden ser configuradas de diferentes formas. Si te vas al enlace “Notification center” de la página de ajustes, verás las diferentes opciones disponibles.

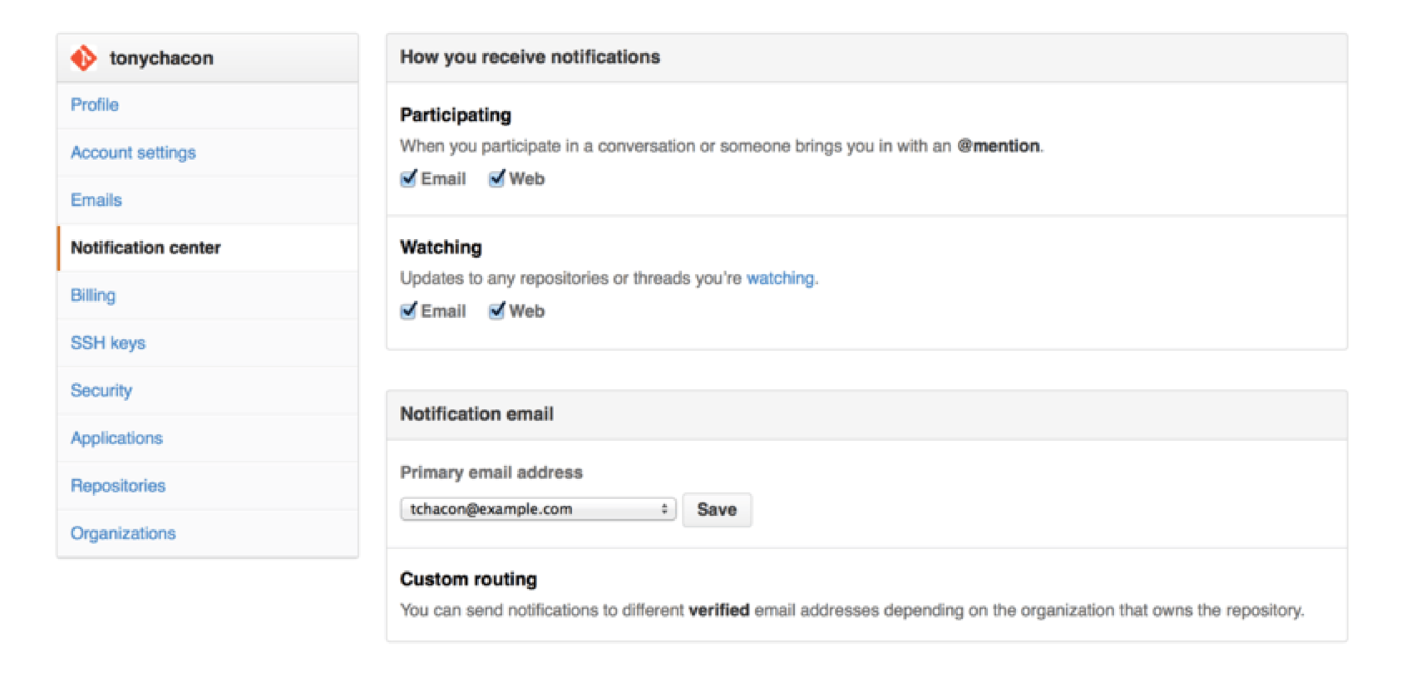


Figure 121. Opciones de Notification center.

Para cada tipo, puedes elegir tener notificaciones de “Email” o de “Web”, y puedes elegir tener una de ellas, ambas o ninguna.

Notificaciones Web

Las notificaciones web se muestran en la página de Github. Si las tienes activas verás un pequeño punto azul sobre el icono de notificaciones en la parte superior de la pantalla, en Centro de notificaciones..

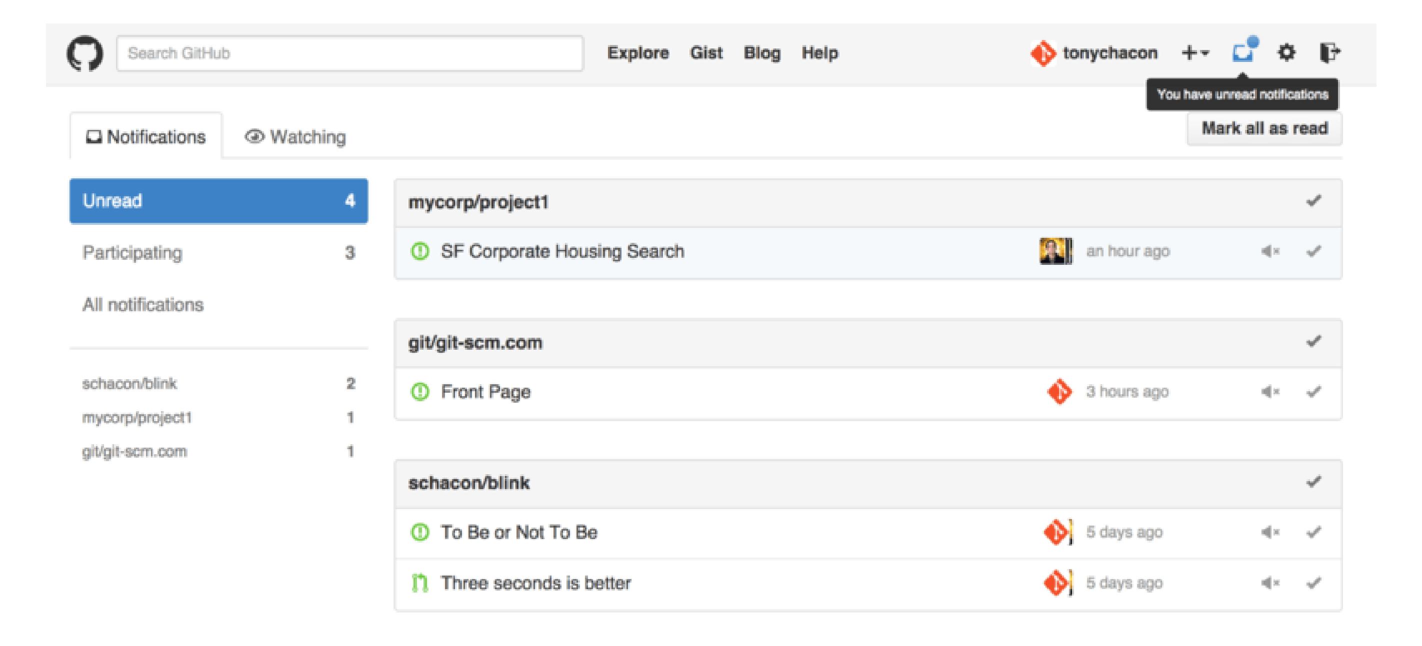


Figure 122. Centro de notificaciones.

Si pulsas en él, verás una lista de todos los elementos sobre los que se te notifica, agrupados por proyecto. Puedes filtrar para un proyecto específico pulsando en el nombre en el lado izquierdo. También puedes reconocer (marcar como leída) una notificación pulsando en el icono de check en una notificación, o reconocerlas *todas* pulsando en el icono de check de todo el grupo. Hay también

un botón “mute” para silenciarlas, que puedes pulsar para no recibir nuevas notificaciones de ese elemento en el futuro.

Todas estas utilidades son útiles para manejar un gran número de notificaciones. Muchos usuarios avanzados de GitHub suelen desactivar las notificaciones por correo y manejarlas todas mediante esta pantalla.

Notificaciones por correo

Las notificaciones por correo electrónico son la otra manera de gestionar notificaciones con GitHub. Si las tienes activa, recibirás los correos de cada notificación. Vimos ya algún ejemplo en Comentarios enviados en notificaciones de correo y Notificación por correo de nuevo Pull Request.. Los correos también serán agrupados correctamente en conversaciones, con lo que estará bien que uses un cliente de correo que maneje las conversaciones.

En las cabeceras de estos correos se incluyen también algunos metadatos, que serán útiles para crear filtros y reglas adecuados.

Por ejemplo, si miramos las cabeceras de los correos enviados a Tony en el correo visto en Notificación por correo de nuevo Pull Request., veremos que se envió la siguiente información:

```
To: tonychacon/fade <fade@noreply.github.com>
Message-ID: <tonychacon/fade/pull/1@github.com>
Subject: [fade] Wait longer to see the dimming effect better (#1)
X-GitHub-Recipient: tonychacon
List-ID: tonychacon/fade <fade.tonychacon.github.com>
List-Archive: https://github.com/tonychacon/fade
List-Post: <mailto:reply+i-4XXX@reply.github.com>
List-Unsubscribe: <mailto:unsub+i-XXX@reply.github.com>,...
X-GitHub-Recipient-Address: tchacon@example.com
```

Vemos en primer lugar que la información de la cabecera `Message-Id` nos da los datos que necesitamos para identificar usuario, proyecto y demás en formato `<usuario>/<proyecto>/<tipo>/<id>`. Si se tratase de una incidencia, la palabra “pull” habría sido reemplazada por “issues”.

Las cabeceras `List-Post` y `List-Unsubscribe` sirven a clientes de correo capaces de interpretarlas ayudarnos a solicitar dejar de recibir nuevas notificaciones de ese tema. Esto es similar a pulsar el botón “mute” que vimos en la versión web, o en “Unsubscribe” en la página de la incidencia o el Pull Request.

También merece la pena señalar que si tienes activadas las notificaciones tanto en la web como por correo, y marcas como leído el correo, en la web también se marcará como leído siempre que permitas las imágenes en el cliente de correo.

Archivos especiales

Hay dos archivos especiales que GitHub detecta y maneja si están presentes en el repositorio.

README

En primer lugar tenemos el archivo `README`, que puede estar en varios formatos. Puede estar con el nombre `README`, `README.md`, `README.asciidoc` y alguno más. Cuando GitHub detecta la presencia del proyecto, lo muestra en la página principal, con el *renderizado* que corresponda a su formato.

En muchos casos este archivo se usa para mostrar información relevante a cualquiera que sea nuevo en el proyecto o repositorio. Esto incluye normalmente cosas como:

- Para qué es el proyecto
- Cómo se configura y se instala
- Ejemplo de uso
- Licencia del código del proyecto
- Cómo participar en su desarrollo

Puesto que GitHub hace un renderizado del archivo, puedes incluir imágenes o enlaces en él para facilitar su comprensión.

CONTRIBUTING

El otro archivo que GitHub reconoce es el archivo `CONTRIBUTING`. Si tienes un archivo con ese nombre y cualquier extensión, GitHub mostrará algo como Apertura de un Pull Request cuando existe el archivo CONTRIBUTING. cuando se intente abrir un Pull Request.

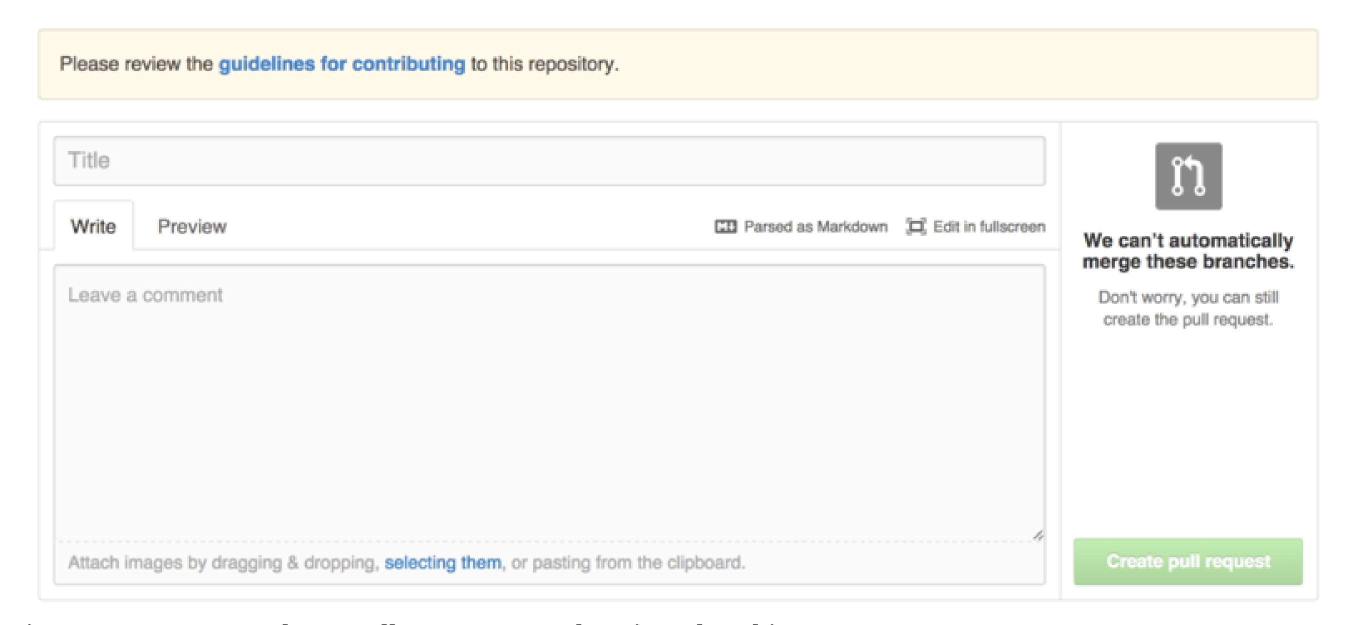


Figure 123. Apertura de un Pull Request cuando existe el archivo CONTRIBUTING.

La idea es que indiques cosas a considerar a la hora de recibir un Pull Request. La gente lo debe leer a modo de guía sobre cómo abrir la petición.

Administración del proyecto

Por lo general, no hay muchas cosas que administrar en un proyecto concreto, pero sí un par de cosas que pueden ser interesantes.

Cambiar la rama predeterminada

Si usas como rama predeterminada una que no sea “master”, por ejemplo para que sea objetivo de los Pull Requests, puedes cambiarla en las opciones de configuración del repositorio, en donde pone “Options”.

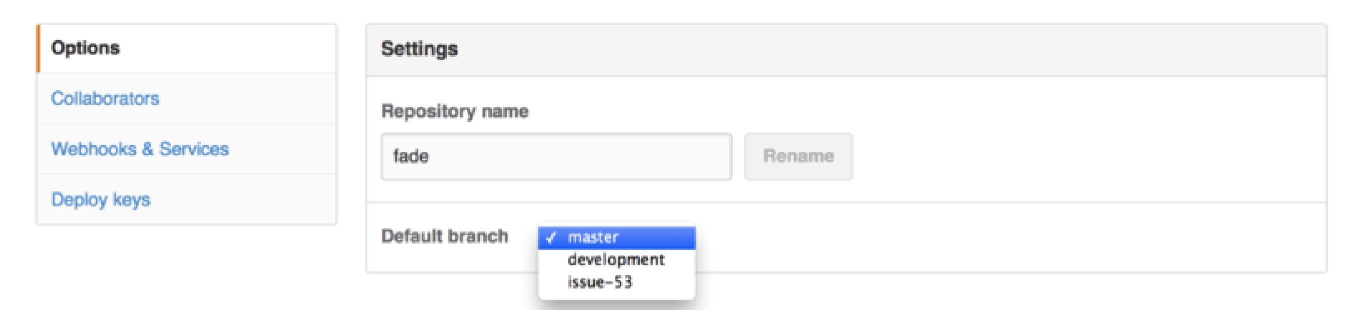


Figure 124. Cambio de la rama predeterminada del proyecto.

Simplemente cambia la rama predeterminada en la lista desplegable, y ésta será la elegida para la mayoría de las operaciones, así mismo será la que sea visible al principio (“checked-out”) cuando alguien clona el repositorio.

Transferencia de un proyecto

Si quieres transferir la propiedad de un proyecto a otro usuario u organización de GitHub, hay una opción para ello al final de “Options” llamada “Transfer ownership”.

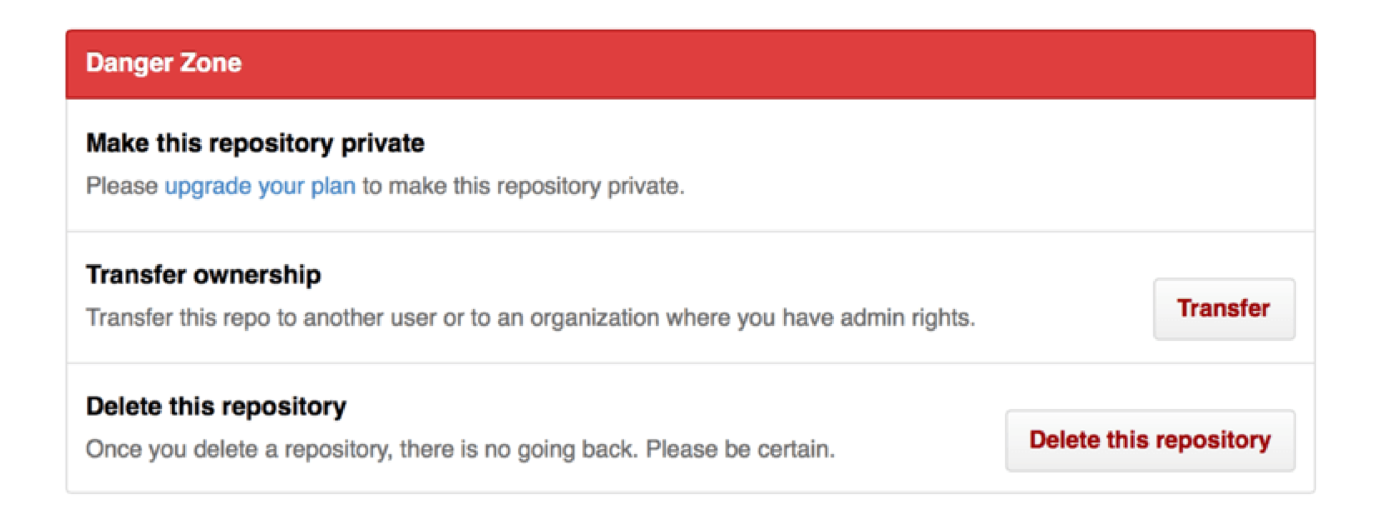


Figure 125. Transferir propiedad de un proyecto.

Esto es útil si vas a abandonar el proyecto y quieres que alguien continúe, o bien se ha vuelto muy grande y prefieres que se gestione desde una organización.

Esta transferencia, supone un cambio de URL. Para evitar que nadie se pierda, genera una redirección web en la URL antigua. Esta redirección funciona también con las operaciones de clonado o de copia desde Git.

Gestión de una organización

Además de las cuentas de usuario, GitHub tiene Organizaciones. Al igual que las cuentas de usuario, las cuentas de organización tienen un espacio donde se guardarán los proyectos, pero en otras cosas son diferentes. Estas cuentas representan un grupo de gente que comparte la propiedad de los proyectos, y además se pueden gestionar estos miembros en subgrupos. Normalmente, estas cuentas se usan en equipos de desarrollo de código abierto (por ejemplo, un grupo para "perl" o para "rails) o empresas (como sería ``google" o "twitter").

Conceptos básicos

Crear una organización es muy fácil: simplemente pulsa en el icono "+" en el lado superior derecho y selecciona "New organization".

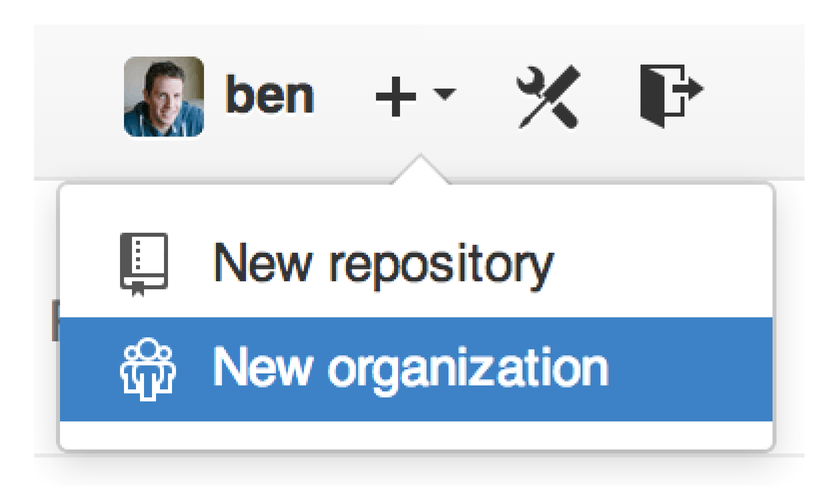


Figure 126. El menú "New organization".

En primer lugar tienes que decidir el nombre de la organización y una dirección de correo que será el punto principal de contacto del grupo. A continuación puedes invitar a otros usuarios a que se unan como co-propietarios de la cuenta.

Sigue estos pasos y serás propietario de un grupo nuevo. Al igual que las cuentas personales, las organizaciones son gratuitas siempre que los repositorios sean de código abierto (y por tanto, públicos).

Como propietario de la organización, cuando bifurcas un repositorio podrás hacerlo a tu elección en el espacio de la organización. Cuando creas nuevos repositorios puedes también elegir el espacio donde se crearán: la organización o tu cuenta personal. Automáticamente, además, quedarás como vigilante (watcher) de los repositorios que crees en la organización.

Al igual que en Tu icono, puedes subir un icono para personalizar un poco la organización, que aparecerá entre otros sitios en la página principal de la misma, que lista todos los repositorios y puede ser vista por cualquiera.

Vamos a ver algunas cosas que son diferentes cuando se hacen con una cuenta de organización.

Equipos

Las organizaciones se asocian con individuos mediante los equipos, que son simplemente

agrupaciones de cuentas de usuario y repositorios dentro de la organización, y qué accesos tienen esas personas sobre cada repositorio.

Por ejemplo, si tu empresa tiene tres repositorios: frontend, backend y deployscripts'; y quieres que los desarrolladores de web tengan acceso a `frontend y tal vez a backend, y las personas de operaciones tengan acceso a backend y deployscripts. Los equipos hacen fácil esta organización, sin tener que gestionar los colaboradores en cada repositorio individual.

La página de la organización te mostrará un panel simple con todos los repositorios, usuarios y equipos que se encuentran en ella.

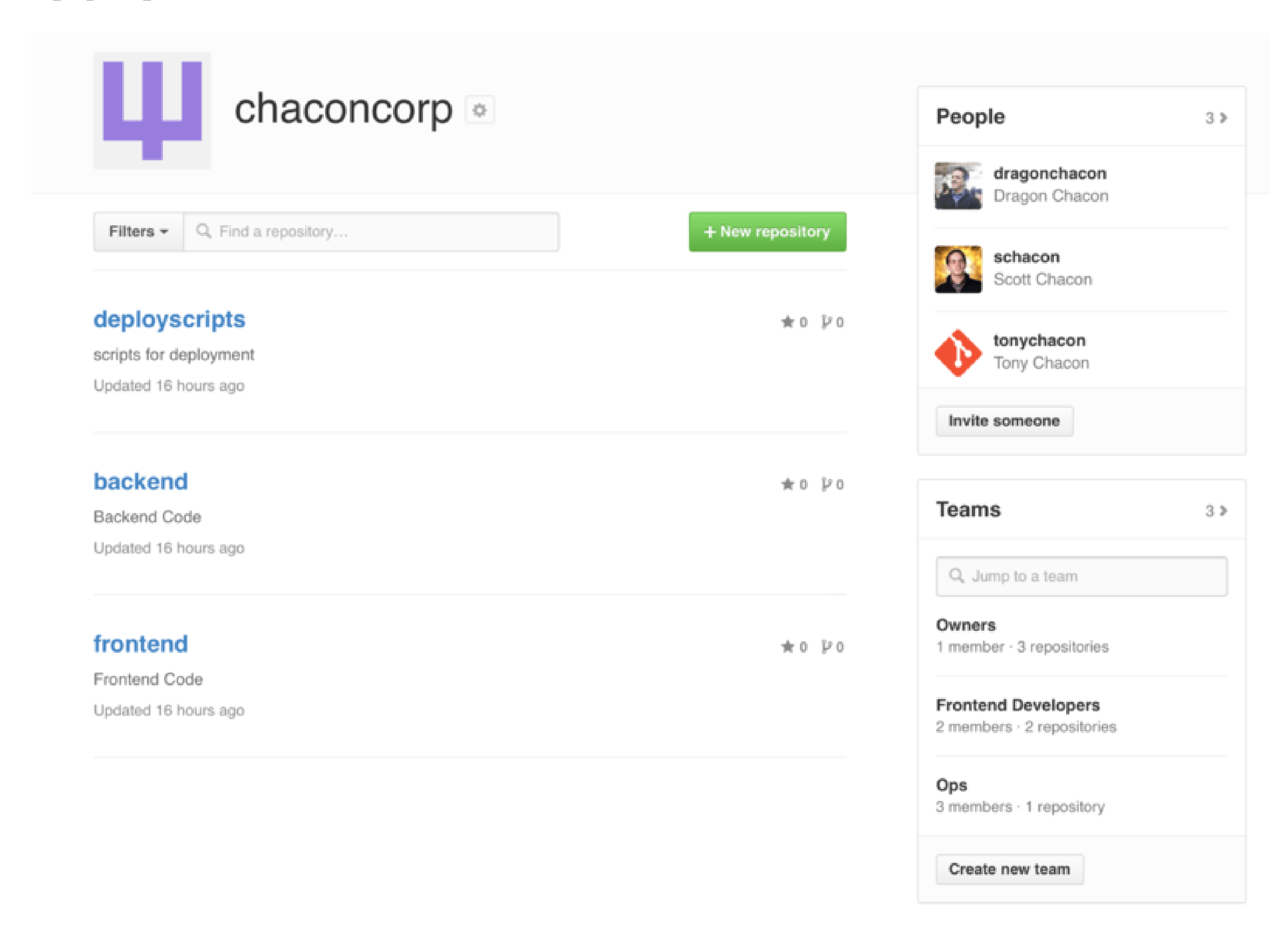


Figure 127. Página de la organización.

Para gestionar tus equipos, puedes pulsar en la barra "Teams" del lado derecho en la página Página de la organización.. Esto te llevará a una página con la que puedes añadir los miembros del equipo, añadir repositorios al equipo o gestionar los ajustes y niveles de acceso del mismo. Cada equipo puede tener acceso sólo lectura, de escritura o administrativo al repositorio. Puedes cambiar el nivel pulsando en el botón "Settings" en Página de equipos..

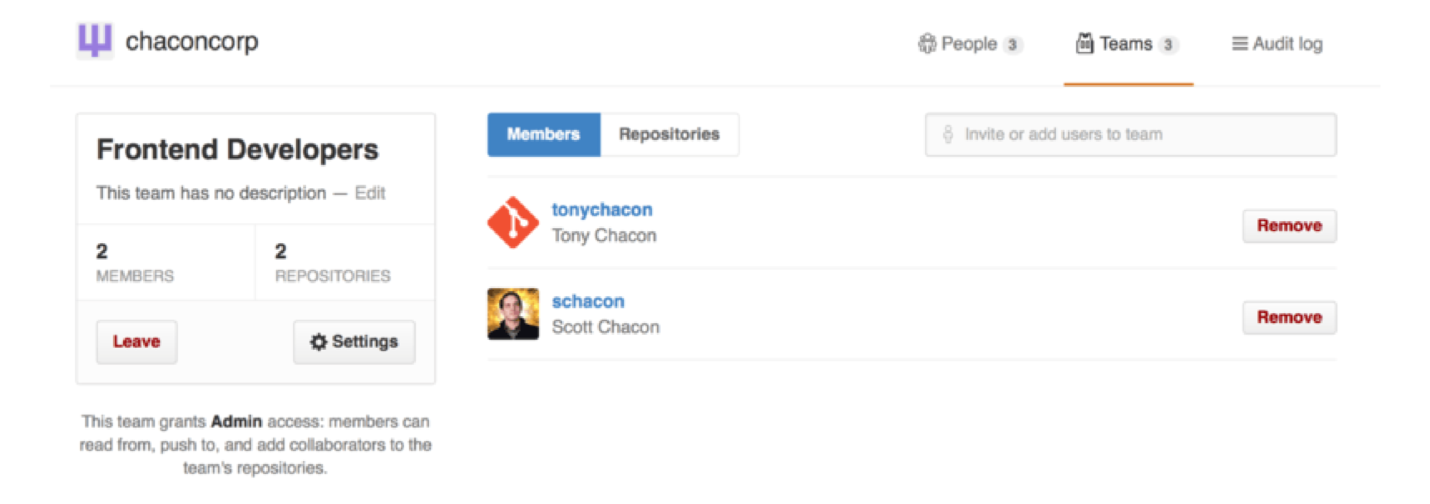


Figure 128. Página de equipos.

Cuando invitas a alguien a un equipo, recibirá un correo con una invitación.

Además, hay menciones de equipo (por ejemplo, `@acmecorp/frontend`) que servirán para que todos los miembros de ese equipo sean suscritos al hilo. Esto resulta útil si quieres involucrar a un equipo en algo al no tener claro a quién concreto preguntar.

Un usuario puede pertenecer a cuantos equipos desee, por lo que no uses equipos solamente para temas de control de acceso a repositorios, sino que puedes usarlo para formar equipos especializados y dispares como `ux`, `css`, `refactoring`, `legal`, etc.

Auditorías

Las organizaciones pueden también dar a los propietarios acceso a toda la información sobre la misma. Puedes incluso ir a la opción *Audit Log* y ver los eventos que han sucedido, quién hizo qué y dónde.

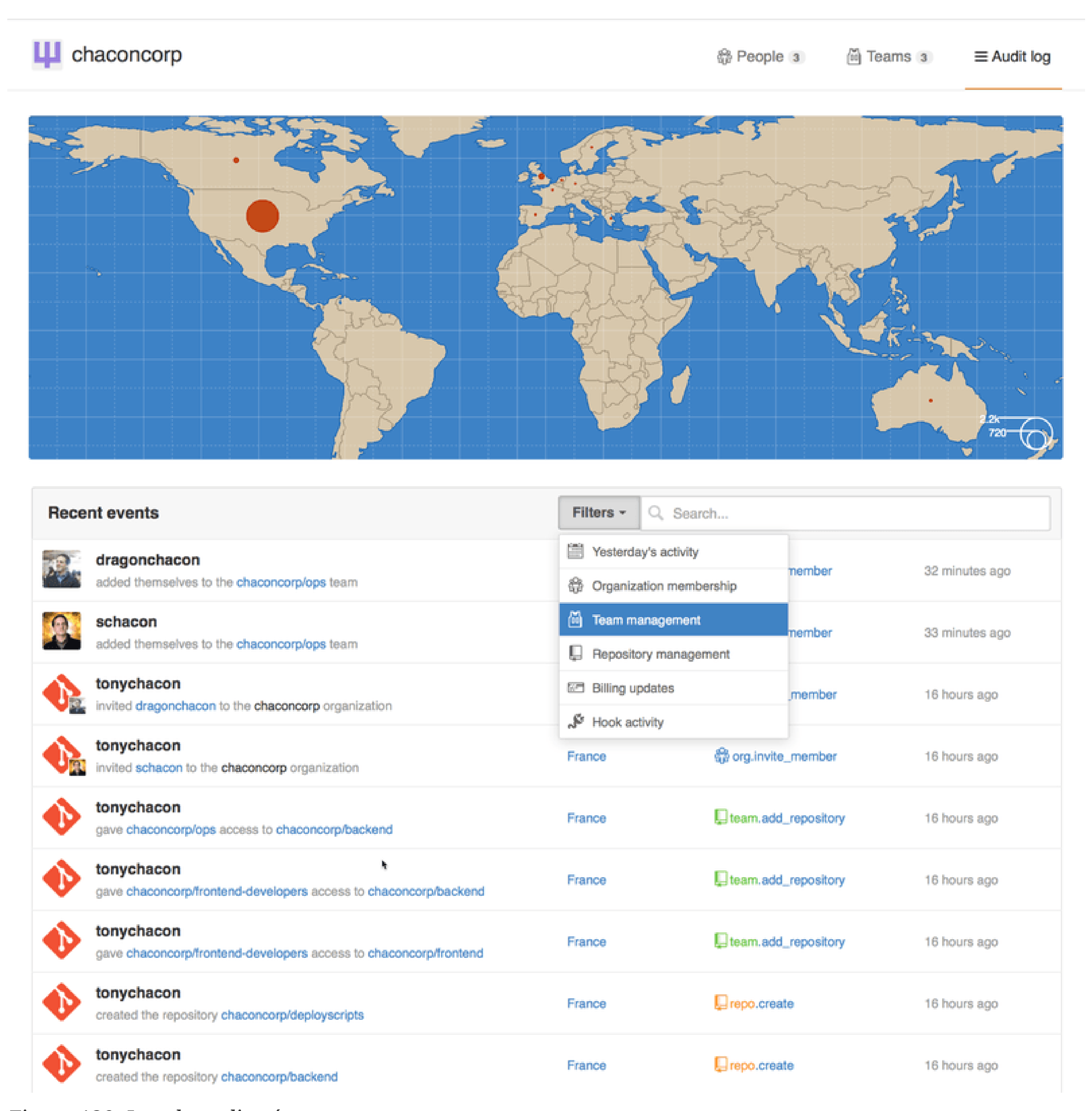


Figure 129. Log de auditoría.

También puedes filtrar por tipo de evento, por lugares o por personas concretas.

Scripting en GitHub

Ya conocemos casi todas las características y modos de trabajo de GitHub. Sin embargo, cualquier grupo o proyecto medianamente grande necesitará personalizar o integrar GitHub con servicios externos.

Por suerte para nosotros, GitHub es bastante *hackeable* en muchos sentidos. En esta sección veremos cómo se usan los *enganches* (hooks) de GitHub y las API para conseguir hacer lo que queremos.

Enganches

Las secciones Hooks y Services de la página de administración del repositorio en Github es la forma más simple de hacer que GitHub interactúe con sistemas externos.

Servicios

En primer lugar, echaremos un ojo a los Servicios. Ambos, enganches y servicios pueden configurarse desde la sección Settings del repositorio, el mismo sitio donde vimos que podíamos añadir colaboradores al proyecto o cambiar la rama predeterminada. Bajo la opción “Webhooks and Services” veremos algo similar a Sección Services and Hooks..

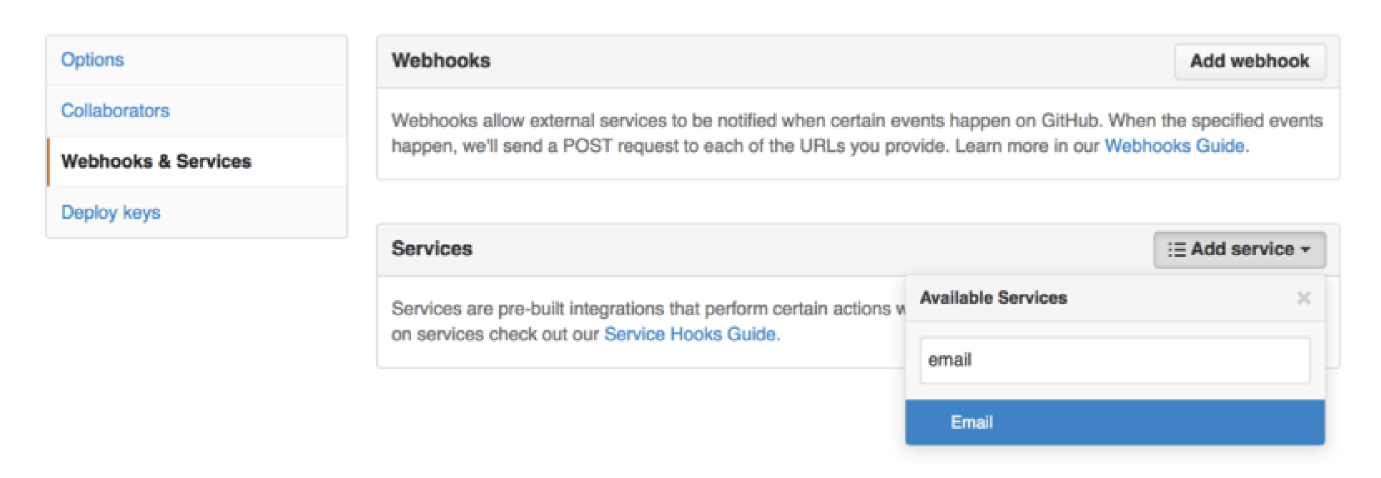


Figure 130. Sección Services and Hooks.

Hay docenas de servicios que podemos elegir, muchos de ellos para integrarse en otros sistemas de código abierto o comerciales. Muchos son servicios de integración continua, gestores de incidencias y fallos, salas de charla y sistemas de documentación. Veremos cómo levantar un servicio sencillo, el enganche con el correo electrónico. Si elegimos “email” en la opción “Add Service” veremos una pantalla de configuración similar a Configuración de servicio de correo..

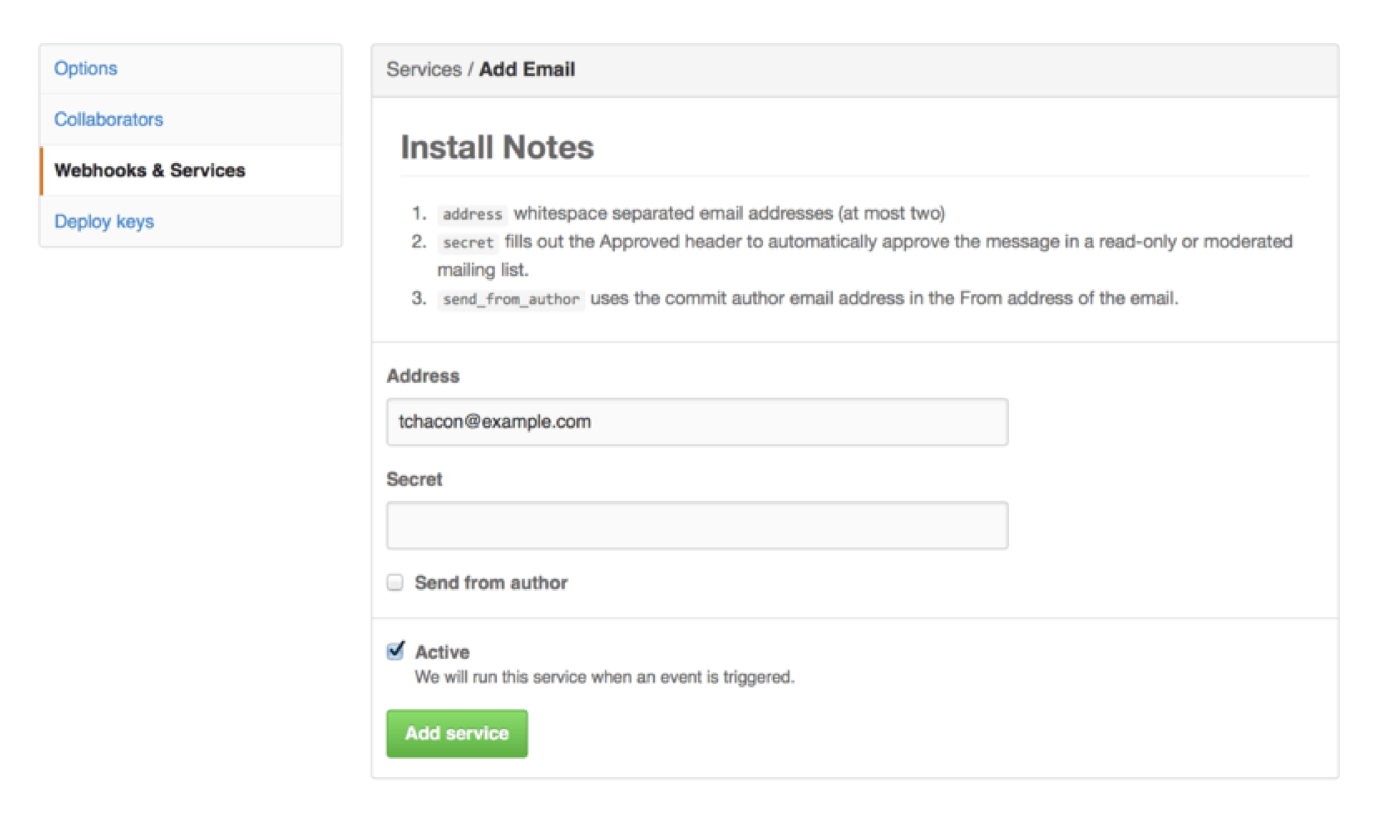


Figure 131. Configuración de servicio de correo.

En este caso, si pulsamos en el botón “Add service”, la dirección de correo especificada recibirá un correo cada vez que alguien envía cambios (push) al repositorio. Los servicios pueden dispararse con muchos otros tipos de eventos, aunque la mayoría solo se usan para los eventos de envío de cambios (push) y hacer algo con los datos del mismo.

Si quieres integrar algún sistema concreto con GitHub, debes mirar si hay algún servicio de integración ya creado. Por ejemplo, si usas Jenkins para ejecutar pruebas de tu código, puedes activar el servicio de integración de Jenkins que lo disparará cada vez que alguien altera el repositorio.

Hooks (enganches)

Si necesitas algo más concreto o quieres integrarlo con un servicio o sitio no incluido en la lista, puedes usar el sistema de enganches más genérico. Los enganches de GitHub son bastante simples. Indicas una URL y GitHub enviará una petición HTTP a dicha URL cada vez que suceda el evento que quieras.

Normalmente, esto funcionará si puedes configurar un pequeño servicio web para escuchar las peticiones de GitHub y luego hacer algo con los datos que son enviados.

Para activar un enganche, pulsa en el botón “Add webhook” de Sección Services and Hooks.. Esto mostrará una página como Configuración de enganches web..

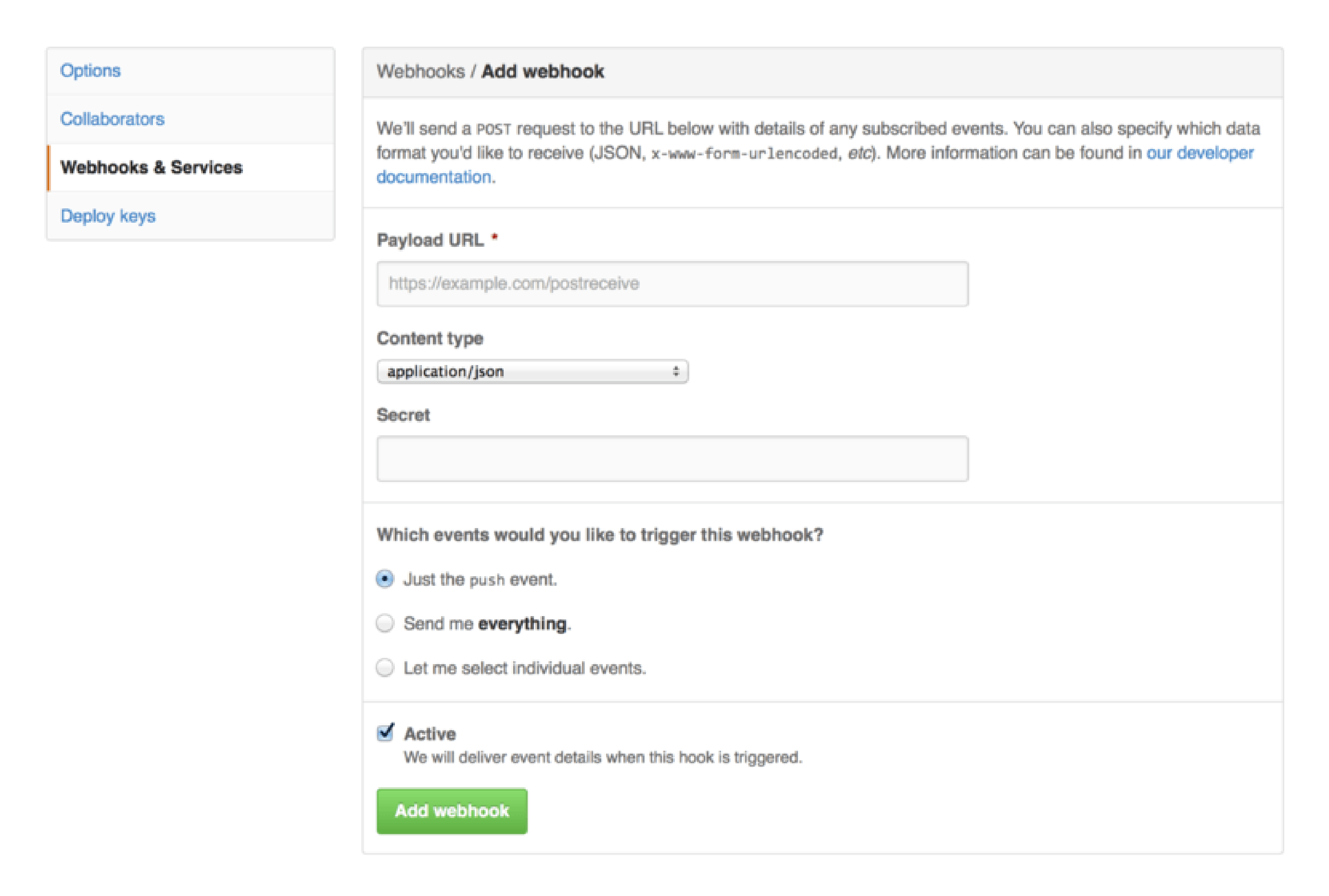


Figure 132. Configuración de enganches web.

La configuración de un enganche web es bastante simple. Casi siempre basta con incluir una URL y una clave secreta, y pulsar en “Add webhook”. Hay algunas opciones sobre qué eventos quieres que disparen el envío de datos (de forma predeterminada el único evento considerado es el evento `push`, de cuando alguien sube algo a cualquier rama del repositorio).

Veamos un pequeño ejemplo de servicio web para manejar un enganche web. Usaremos el entorno Sinatra de Ruby puesto que es conciso y podrás entender con facilidad qué estamos haciendo.

Pongamos que queremos recibir un correo cada vez que alguien sube algo a una rama concreta del repositorio, modificando un archivo en particular. Podríamos hacerlo con un código similar a este:

```
require 'sinatra'
require 'json'
require 'mail'

post '/payload' do
  push = JSON.parse(request.body.read) # parse the JSON

  # gather the data we're looking for
  pusher = push["pusher"]["name"]
  branch = push["ref"]

  # get a list of all the files touched
  files = push["commits"].map do |commit|
    commit['added'] + commit['modified'] + commit['removed']
  end
  files = files.flatten.uniq

  # check for our criteria
  if pusher == 'schacon' &&
     branch == 'ref/heads/special-branch' &&
     files.include?('special-file.txt')

    Mail.deliver do
      from     'tchacon@example.com'
      to       'tchacon@example.com'
      subject  'Scott Changed the File'
      body     "ALARM"
    end
  end
end
```

Aquí estamos tomando el bloque JSON que GitHub entrega y mirando quién hizo el envío, qué rama se envió y qué archivos se modificaron en cada commit realizado en este push. Entonces, comprobamos si se cumple nuestro criterio y enviamos un correo si es así.

Para poder probar algo como esto, tienes una consola de desarrollador en la misma pantalla donde configuraste el enganche, donde se puede ver las últimas veces que GitHub ha intentado ejecutar el enganche. Para cada uno, puedes mirar qué información se ha enviado, y si fue recibido correctamente, junto con las cabeceras correspondientes de la petición y de la respuesta. Esto facilita mucho las pruebas de tus enganches.

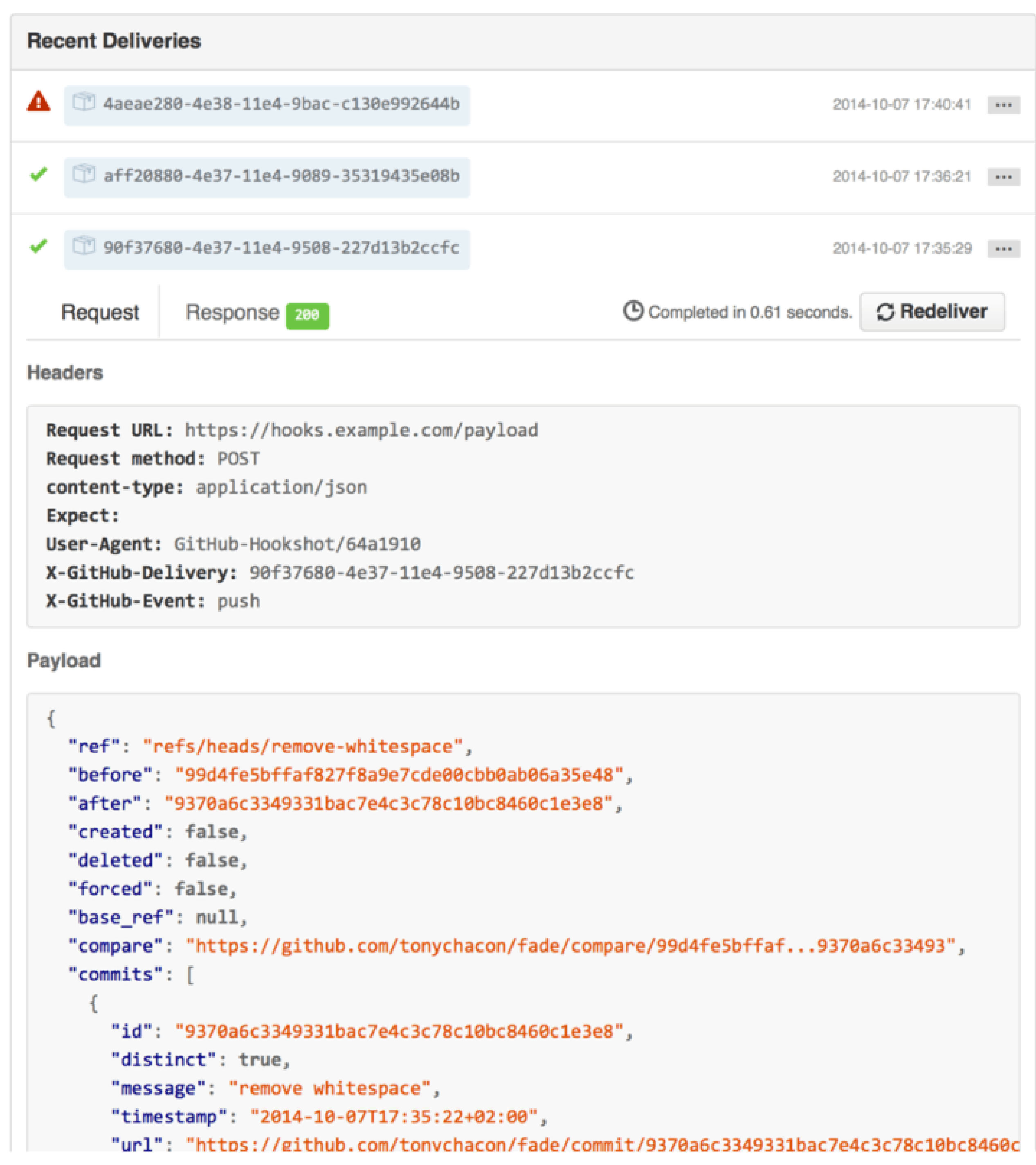


Figure 133. Depuración de un web hook.

Otra cosa muy interesante es que puedes repetir el envío de cualquier petición para probar el servicio con facilidad.

Para más información sobre cómo escribir webhooks (enganches) y los diferentes tipos de eventos que puedes tratar, puedes ir a la documentación del desarrollador de GitHub, en: https://developer.github.com/webhooks/

La API de GitHub

Servicios y enganches nos sirven para recibir notificaciones “push” sobre eventos que suceden en tus repositorios. Pero, ¿qué pasa si necesitas más información acerca de estos eventos? ¿y si necesitas automatizar algo como añadir colaboradores o etiquetar incidencias?

Aquí es donde entra en juego la API de GitHub. GitHub tiene montones de llamadas de API para hacer casi cualquier cosa que puedes hacer via web, de forma automatizada. En esta sección aprenderemos cómo autentificar y conectar a la API, cómo comentar una incidencia y cómo cambiar el estado de un Pull Request mediante la API.

Uso Básico

Lo más básico que podemos hacer es una petición GET a una llamada que no necesite autentificación. Por ejemplo, información solo lectura de un proyecto de código abierto. Por ejemplo, si queremos conocer información acerca del usuario "schacon", podemos ejecutar algo como:

```
$ curl https://api.github.com/users/schacon
{
  "login": "schacon",
  "id": 70,
  "avatar_url": "https://avatars.githubusercontent.com/u/70",
# …
  "name": "Scott Chacon",
  "company": "GitHub",
  "following": 19,
  "created_at": "2008-01-27T17:19:28Z",
  "updated_at": "2014-06-10T02:37:23Z"
}
```

Hay muchísimas llamadas como esta para obtener información sobre organizaciones, proyectos, incidencias, commits, es decir, todo lo que podemos ver públicamente en la web de GitHub. Se puede usar la API para otras cosas como ver un archivo Markdown cualquier o encontrar una plantilla de `.gitignore`.

```
$ curl https://api.github.com/gitignore/templates/Java
{
  "name": "Java",
  "source": "*.class

# Mobile Tools for Java (J2ME)
.mtj.tmp/

# Package Files #
*.jar
*.war
*.ear

# virtual machine crash logs, see
http://www.java.com/en/download/help/error_hotspot.xml
hs_err_pid*
"
}
```

Comentarios en una incidencia

Sin embargo, si lo que quieres es realizar una acción como comentar una incidencia o un Pull Request, o si quieres ver o interactuar con un contenido privado, necesitas identificarte.

Hay varias formas de hacerlo. Puedes usar la autentificación básica, con tu usuario y tu contraseña, aunque generalmente es mejor usar un token de acceso personal. Puedes generarlo en la opción "Applications" de tu página de ajustes personales.

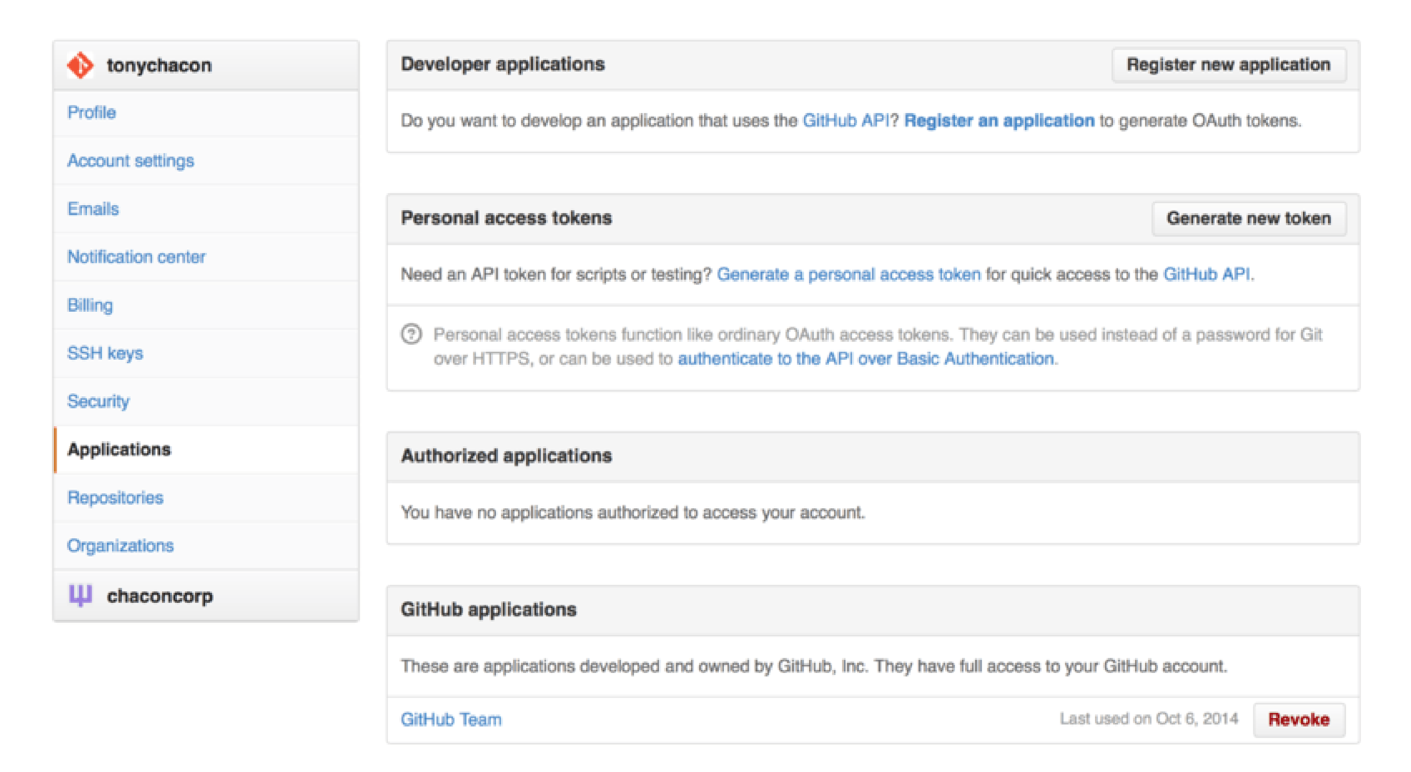


Figure 134. Generación del token de acceso.

Te preguntará acerca del ámbito que quieres para el token y una descripción. Asegúrate de usar una buena descripción para que te resulte fácil localizar aquellos token que ya no necesitas.

GitHub te permitirá ver el token una vez, por lo que tienes que copiarlo en ese momento. Ahora podrás identificarte en el script con el token, en lugar del usuario y la contraseña. Esto está bien porque puedes limitar el ámbito de lo que se quiere hacer y porque el token se puede anular.

También tiene la ventana de incrementar la tasa de accesos. Sin la autentificación podrás hacer 60 peticiones a la hora. Con una identificación el número de accesos permitidos sube a 5,000 por hora.

Realicemos entonces un comentario en una de nuestras incidencias. Por ejemplo, queremos dejar un comentario en la incidencia #6. Para ello, hacemos una petición HTTP POST a `repos/<usuario>/<repo>/issues/<num>/comments` con el token que acabamos de generar como cabecera Authorization.

```
$ curl -H "Content-Type: application/json" \
       -H "Authorization: token TOKEN" \
       --data '{"body":"A new comment, :+1:"}' \
       https://api.github.com/repos/schacon/blink/issues/6/comments
{
  "id": 58322100,
  "html_url": "https://github.com/schacon/blink/issues/6#issuecomment-58322100",
  ...
  "user": {
    "login": "tonychacon",
    "id": 7874698,
    "avatar_url": "https://avatars.githubusercontent.com/u/7874698?v=2",
    "type": "User",
  },
  "created_at": "2014-10-08T07:48:19Z",
  "updated_at": "2014-10-08T07:48:19Z",
  "body": "A new comment, :+1:"
}
```

Ahora, si vas a la incidencia, verás el comentario que acabas de enviar tal como en Comentario enviado desde la API de GitHub..

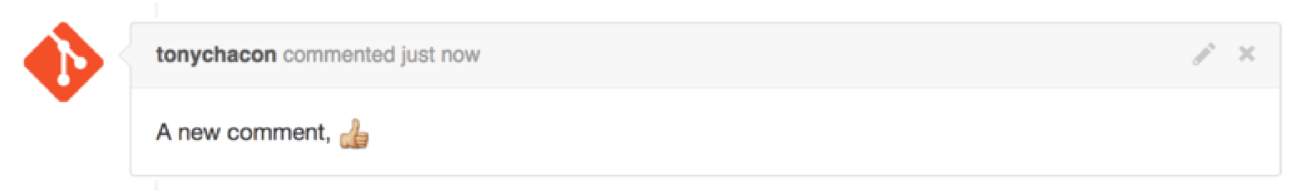


Figure 135. Comentario enviado desde la API de GitHub.

Puedes usar la API para hacer todo lo que harías en el sitio web: crear y ajustar hitos, asignar gente a incidencias o pull requests, crear y cambiar etiquetas, acceder a datos de commit, crear nuevos commits y ramas, abrir, cerrar o fusionar Pull Requests, crear y editar equipos, comentar líneas de cambio en Pull Requests, buscar en el sitio y mucho más.

Cambio de estado de un Pull Request

Un ejemplo final que veremos es realmente útil si trabajas con Pull Requests. Cada commit tiene uno o más estados asociados con ellos, y hay una API para alterar y consultar ese estado.

Los servicios de integración continua y pruebas hacen uso de esta API para actuar cuando alguien envía código al repositorio, probando el mismo y devolviendo como resultado si el commit pasó todas las pruebas. Además, se podría comprobar si el mensaje del commit tiene un formato adecuado, si el autor siguió todas las recomendaciones para autores, si fue firmado, etc.

Supongamos que tenemos un enganche web en el repositorio que llama a un servicio web que comprueba si en el mensaje del commit aparece la cadena `Signed-off-by`.

```ruby
require 'httparty'
require 'sinatra'
require 'json'

post '/payload' do
  push = JSON.parse(request.body.read) # parse the JSON
  repo_name = push['repository']['full_name']

  # look through each commit message
  push["commits"].each do |commit|

    # look for a Signed-off-by string
    if /Signed-off-by/.match commit['message']
      state = 'success'
      description = 'Successfully signed off!'
    else
      state = 'failure'
      description = 'No signoff found.'
    end

    # post status to GitHub
    sha = commit["id"]
    status_url = "https://api.github.com/repos/#{repo_name}/statuses/#{sha}"

    status = {
      "state"       => state,
      "description" => description,
      "target_url"  => "http://example.com/how-to-signoff",
      "context"     => "validate/signoff"
    }
    HTTParty.post(status_url,
      :body => status.to_json,
      :headers => {
        'Content-Type'  => 'application/json',
        'User-Agent'    => 'tonychacon/signoff',
        'Authorization' => "token #{ENV['TOKEN']}" }
    )
  end
end
```

Creemos que esto es fácil de seguir. En este controlador del enganche, miramos en cada commit enviado, y buscamos la cadena *Signed-off-by* en el mensaje de commit, y finalmente hacemos un HTTP POST al servicio de API `/repos/<user>/<repo>/statuses/<commit_sha>` con el resultado.

En este caso, puedes enviar un estado (*success*, *failure*, *error*), una descripción de qué ocurrió, un URL objetivo donde el usuario puede ir a buscar más información y un "contexto" en caso de que haya múltiples estados para un commit. Por ejemplo, un servicio de test puede dar un estado, y un servicio de validación puede dar por su parte su propio estado; el campo "context" serviría para diferenciarlos.

Si alguien abre un nuevo Pull Request en GitHub y este enganche está configurado, verías algo como Estado del commit mediante API..

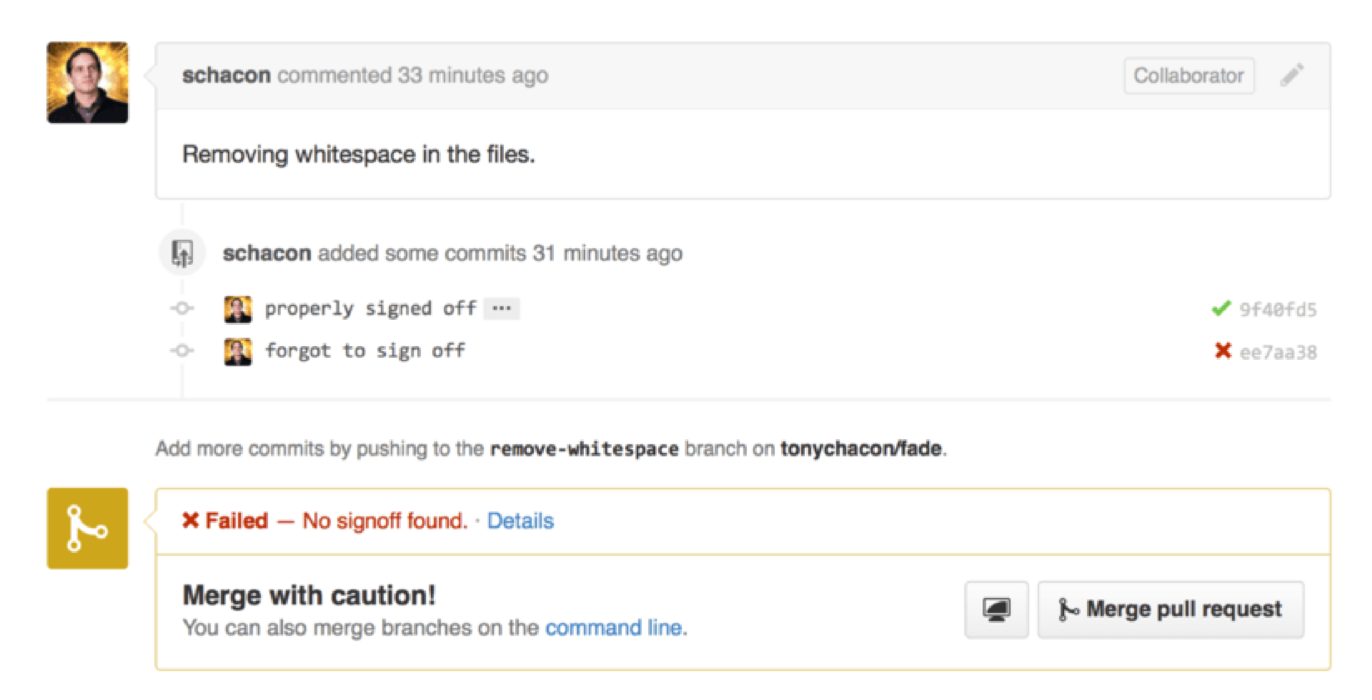


Figure 136. Estado del commit mediante API.

Podrás ver entonces una pequeña marca de color verde, que nos indica que el commit tiene la cadena "Signed-off-by" en el mensaje y un aspa roja en aquellos donde el autor olvidase hacer esa firma. También verías que el Pull Request toma el estado del último commit en la rama y te avisa de si es un fallo. Esto es realmente útil si usas la API para pruebas, y así evitar hacer una fusión accidental de unos commits que han fallado las pruebas.

Octokit

Hasta ahora hemos hecho casi todas las pruebas con `curl` y peticiones HTTP simples, pero en GitHub hay diferentes bibliotecas de código abierto que hacen más fácil el manejo de la API, agrupadas bajo el nombre de Octokit. En el momento de escribir esto, están soportados lenguajes como Go, Objective-C, Ruby y .NET. Se puede ir a http://github.com/octokit para más información sobre esto, que te ayudarán a manejar peticiones y respuestas a la API de GitHub.

Con suerte estas utilidades te ayudarán a personalizar y modificar GitHub para trabajar mejor con tu forma concreta de trabajar. Para una documentación completa de la API así como ayudas para realizar tareas comunes, puedes consultar en https://developer.github.com.

Resumen

Ahora ya eres un usuario de GitHub. Ya sabes cómo crear una cuenta, gestionar una organización, crear y enviar repositorios, participar con los proyectos de otras personas y aceptar contribuciones de terceros en tus proyectos. En el siguiente capítulo conoceremos otras herramientas y trucos potentes para manejar situaciones más complicadas, con los que te puedes convertir con seguridad en un experto de Git.

Herramientas de Git

Hasta ahora, ya has aprendido la mayoría de los comandos diarios y el flujo de trabajo que necesitas para manejar y mantener un repositorio de Git para tu control del código fuente. Has conseguido cumplir con las tareas básicas de seguimiento y has agregado archivos, además has aprovechado el poder del area de staging y has conocido el tema de branching y merging.

Ahora vas a explorar unas cuantas cosas bastantes poderosas que Git puede realizar y que no necesariamente vas a usar en tu día a día, pero que puedes necesitar en algún momento.

Revisión por selección

Git te permite especificar ciertos commits o un rango de estos en muchas maneras. No son necesariamente obvias, pero es útil conocerlas.

Revisiones individuales

Obviamente se puede referir a un commit por el hash SHA-1 que se le asigna, pero también existen formas más amigables de referirse a los commits. Esta sección delinea varias maneras en las que se puede referir a un commit indiviual.

SHA-1 corto

Git es lo suficientemente inteligente como para descifrar el commit al que te refieres si le entregas los primeros caracteres, siempre y cuando la parte de SHA-1 sea de al menos 4 caracteres y no sea ambigua - esto quiere decir, que solamente un objeto en el repositorio actual comience con ese SHA-1 parcial.

Por ejemplo, para ver un commit específico, supongamos que se utiliza el comando `git log` y se identifica el commit donde se agregó cierta funcionalidad:

```
$ git log
commit 734713bc047d87bf7eac9674765ae793478c50d3
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'

commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800

    added some blame and merge stuff
```

En este caso, se escoge `1c002dd....`. Si usted utiliza `git show` en ese commit, los siguientes comandos son iguales (asumiendo que las versiones cortas no son ambiguas):

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
$ git show 1c002dd4b536e7479f
$ git show 1c002d
```

Git puede descubrir una abreviación corta y única del valor SHA-1. Si usted añade `--abbrev-commit` al comando `git log`, el resultado utilizará los valores cortos pero manteniendolos únicos; por default utiliza siete caracteres pero los hace más largos de ser necesario para mantener el SHA-1 sin ambigüedades:

```
$ git log --abbrev-commit --pretty=oneline
ca82a6d changed the version number
085bb3b removed unnecessary test code
a11bef0 first commit
```

Generalmente, de ocho a diez caracteres son más que suficientes para ser únicos en un proyecto.

Como un ejemplo, el kernel Linux, que es un proyecto bastante grande con alrededor de 450 mil commits y 3.6 millones de objetos, no tiene dos objetos cuyos SHA-1s se superpongan antes de los primeros 11 caracteres.

NOTE

UNA BREVE NOTE RESPECTO A SHA-1

Mucha gente se preocupa de que en cierto momento, fruto del azar, tendrán dos objetos en us repositorio cuyos hash tendrán el mismo valor SHA-1. Pero, ¿entonces qué?

Si sucede que realizas un commit y el objeto tiene el mismo hash que un objeto previo en tu repositorio, Git verá el objeto previo en tu base de datos y asumirá que ya estaba escrito. Si intentas mirar el objeto otra vez, siempre tendrás la data del primer objeto.

Sin embargo, debes ser consciente de cuán ridículamente improbable es este escenario. El digest SHA-1 es de 20 bytes o 160 bits. El número de objetos aleatorios necesario para asegurar un 50% de probabilidades de una única colisión bordea el 2^{80} (la fórmula para determinar la propabilidad de colisión es `p = (n(n-1)/2) * (1/2^160))`. 2^{80} es 1.2 x 10^{24} o 1 millón de millones de millones. Esto es 1,200 veces el ńumero de granos de arena en la Tierra.

Aquí hay un ejemplo para darte una idea de lo que tomaría para conseguir una colisión de SHA-1. Si todos los 6.5 mil millones de humanos en la Tierra estuvieran programando, y cada segundo, cada uno produjera el código equivalente a toda la historia del kernel Linux (3.6 mil millones de objetos en Git) e hicieran push en un enorme repositorio Git, tomaría aproximadamente 2 años hasta que el repositorio tuviera suficientes objetos para un 50% de probabilidades de que ocurriera una única colisión en los SHA-1 de los objetos. Existe una probabilidad más alta de que cada miembro de tu equipo de programación sea atacado y asesinado por lobos en incidentes sin relación y todo esto en la misma noche.

Referencias por rama

El camino más sencillo para especificar un commit requiere que este tenga una rama de referencia apuntando al mismo. Entonces, se puede usar el nombre de la rama en cualquier comando Git que espere un objeto commit o un valor SHA-1. Por ejemplo, si se quiere mostrar el último objeto commit de una rama, los siguientes comandos son equivalentes, asumiendo que la rama `topic1` apunta a `ca82a6d`:

```
$ git show ca82a6dff817ec66f44342007202690a93763949
$ git show topic1
```

Si se quiere ver a qué SHA-1 apunta un rama en específico, o si se quiere ver lo que cualquiera de estos ejemplos expresa en terminos de SHA-1s, puede utilizar una herramienta de plomería de Git llamada `rev-parse`. Se puede ver Los entresijos internos de Git para más información sobre las herramientas de plomería; básicamente, `rev-parse` existe para operaciones de bajo nivel y no está diseñado para ser utilizado en operaciones diarias. Aquí puedes correr `rev-parse` en tu rama.

```
$ git rev-parse topic1
ca82a6dff817ec66f44342007202690a93763949
```

Nombres cortos de RefLog

Una de las cosas que Git hace en segundo plano mientras tu estás trabajando a distancia es mantener un “reflog” - un log de a dónde se apuntan las referencias de tu HEAD y tu rama en los últimos meses.

Se puede ver el reflog utilizando `git reflog`:

```
$ git reflog
734713b HEAD@{0}: commit: fixed refs handling, added gc auto, updated
d921970 HEAD@{1}: merge phedders/rdocs: Merge made by recursive.
1c002dd HEAD@{2}: commit: added some blame and merge stuff
1c36188 HEAD@{3}: rebase -i (squash): updating HEAD
95df984 HEAD@{4}: commit: # This is a combination of two commits.
1c36188 HEAD@{5}: rebase -i (squash): updating HEAD
7e05da5 HEAD@{6}: rebase -i (pick): updating HEAD
```

Cada vez que la punta de tu rama es actualizada por cualquier razón, Git guarda esa información en este historial temporal. Y es así como se puede especificar commits antiguos con esta información. Si se quiere ver el quinto valor anterior a tu HEAD en el repositorio, se puede usar la referencia `@{n}` que se ve en la salida de reflog:

```
$ git show HEAD@{5}
```

También se puede utilizar esta sintaxis para ver dónde se encontraba una rama dada una cierta cantidad de tiempo. Por ejemplo, para ver dónde se encontraba tu rama `master` ayer, se puede utilizar

```
$ git show master@{yesterday}
```

Esto muestra a dónde apuntaba tu rama el día de ayer. Esta técnica solo funciona para información que permanece en tu reflog, por lo que no se puede utilizar para ver commits que son anteriores a los que aparecen en él.

Para ver información sobre reflog en el formato de `git log`, se puede utilizar `git log -g`:

```
$ git log -g master
commit 734713bc047d87bf7eac9674765ae793478c50d3
Reflog: master@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: commit: fixed refs handling, added gc auto, updated
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Reflog: master@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: merge phedders/rdocs: Merge made by recursive.
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'
```

Es importante notar que la información de reflog es estríctamente local - es un log de lo que se ha hecho en el repositorio local. Las referencias no serán las mismas en otra copia del repositorio; y justo después de que se ha inicializado el repositorio, se tendrá un reflog vacío, dado que no ha ocurrido ninguna actividad todavía en el mismo. Utilizar `git show HEAD@{2.months.ago}` funcionará solo si se clonó el projecto hace al menos dos meses - si se clonó hace cinco minutos, no se obtendrán resultados.

Referencias por ancestros

Otra forma principal de especificar un commit es por sus ancestros. Si se coloca un `^` al final de la referencia, Git lo resuelve como el padre de ese commit. Supongamos que se mira a la historia de un proyecto:

```
$ git log --pretty=format:'%h %s' --graph
* 734713b fixed refs handling, added gc auto, updated tests
*   d921970 Merge commit 'phedders/rdocs'
|\
| * 35cfb2b Some rdoc changes
* | 1c002dd added some blame and merge stuff
|/
* 1c36188 ignore *.gem
* 9b29157 add open3_detach to gemspec file list
```

Entoces, se puede ver los commits previos especificando `HEAD^`, lo que significa “el padre de HEAD”:

```
$ git show HEAD^
commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'
```

También se puede especificar un número despúes de `^` - por ejemplo, `d921970^2` significa “el segundo padre de d921970.” Esta sintaxis es útil solamente para fusiones confirmadas, las cuales tienen más de un padre. El primer padre es la rama en el que se estaba al momento de fusionar, y el segundo es el commit en la rama en la que se fusionó:

```
$ git show d921970^
commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800

    added some blame and merge stuff

$ git show d921970^2
commit 35cfb2b795a55793d7cc56a6cc2060b4bb732548
Author: Paul Hedderly <paul+git@mjr.org>
Date:   Wed Dec 10 22:22:03 2008 +0000

    Some rdoc changes
```

La otra manera principal de especificar ancestros es el `~`. Este también refiere al primer padre, asi que `HEAD~` y `HEAD^` son equivalentes. La diferencia se vuelve aparente cuando se especifica un número. `HEAD~2` significa “el primer padre del primer padre,” o “el abuelo” - este recorre el primer padre las veces que se especifiquen. Por ejemplo, en el historial listado antes, `HEAD~3` sería

```
$ git show HEAD~3
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

    ignore *.gem
```

Esto también puede ser escrito `HEAD^^^`, lo que también es, el primer padre del primer padre del primer padre:

```
$ git show HEAD^^^
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

    ignore *.gem
```

También se puede combinar estas sintaxis - se puede obtener el segundo padre de la referencia previa (asumiendo que fue una fusión confirmada) utilizando `HEAD~3^2`, y así sucesivamente.

Rangos de Commits

Ahora que ya puede especificar commits individuales, vamos a a ver cómo especificar un rango de commits. Esto es particularmete útil para administrar las ramas - si se tienen muchas ramas, se puede usar un rango de especificaciones para contestar preguntas como, “¿Qué trabajo está en esta rama y cuál no hemos fusionado en la rama principal?”

Dos puntos

La forma más común de especificar un rango es mediante la sintaxis de doble punto. Esto básicamente pide a Git que resuelva un rango de commits que es alcanzable desde un commit pero que no es alcanzable desde otro. Por ejemplo, digamos que se tiene un historial de commits que se ve como Example history for range selection..

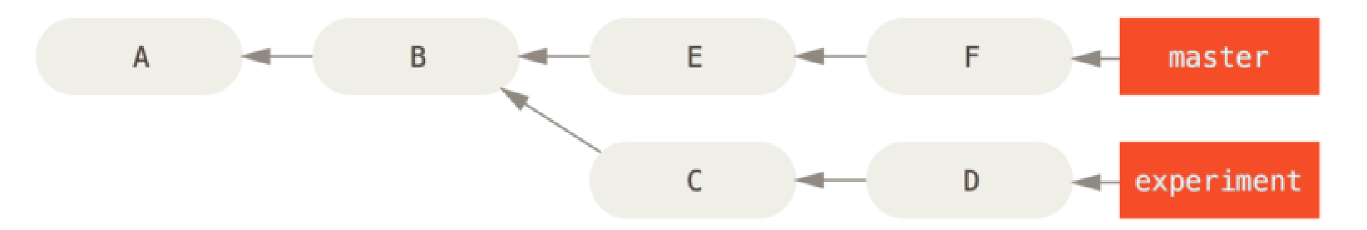


Figure 137. Example history for range selection.

Se quiere ver qué se encuentra en la rama experiment que no ha sido fusionado a la rama master todavía. Se puede pedir a Git que muestre el log de solamente aquellos commits con `master..experiment` - eso significa “todos los commits alcanzables por experiment que no son alcanzables por master.” Para ser breves y claros en este ejemplo. Se usarán las letras de los objetos commit del diagrama en lugar del log para que se muestre de la siguiente manera:

```
$ git log master..experiment
D
C
```

Si, por otro lado, se quiere lo opuesto - todos los commits en `master` que no están en `experiment` - se pueden invertir los nombres de las ramas. `experiment..mater``muestra todo lo que hay en `master` que no es alcanzable para `experiment`:

```
$ git log experiment..master
F
E
```

Esto es útil si se quiere mantener la rama `experiment` actualizada y previsualizar lo que se está a punto de fusionar. Otro uso bastante frecuente de esta sintaxis es para ver lo que se está a punto de publicar en remote:

```
$ git log origin/master..HEAD
```

Este comando muestra cualquier commit en tu rama actual que no está en la rama `master` del remoto `origin`. Si se corre un `git push` y la rama de seguimiento actual es `origin/master`, los commits listados por `git log origin/master..HEAD` son los commits que serán transferidos al servidor. También se puede dejar de lado de la sintaxis para que Git asuma HEAD. Por ejemplo, se puede obtener el mismo resultado que en el ejemplo previo tipiando `git log origin/master..` - Git sustituye HEAD is un lado está faltando.

Múltiples puntos

La sintaxis de dos puntos es útil como una abreviatura; pero tal vez se desea especificar más de dos ramas para indicar la revisión, como puede ser revisar qué commits existen en muchas ramas que no se encuentran en la rama en la que se realiza el trabajo actualmente. Git permite realizar esto utilizando el caracter `^` or `--not` antes de cualquier referencia de la cual no deseas ver los commits alcanzables.

```
$ git log refA..refB
$ git log ^refA refB
$ git log refB --not refA
```

Esto es bueno porque con esta sintaxis se puede especificar más de dos referencias en una consulta, lo que no se puede hacer con la sintaxis de dos puntos. Por ejemplo, si se quiere ver todos los commits que son alcanzables desde `refA` o `refB` pero no desde `refC`, se puede escribir lo siguiente:

```
$ git log refA refB ^refC
$ git log refA refB --not refC
```

Esto lo convierte en un sistema de consultas muy poderoso que debería ayudar a descubrir qué hay en sus ramas.

Tres puntos

La última sintaxis de selección de rangos es la de tres puntos, que especifica todos los commits que son alcanzables por alguna de dos referencias pero no por las dos al mismo tiempo. Mire atrás al ejemplo de historial de commits en Example history for range selection.. Si se quiere ver lo que está en `master` o `experiment` pero no en ambos, se puede utilizar

```
$ git log master...experiment
F
E
D
C
```

Nuevamente, esto entrega la salida normal del `log` pero muestra solo la información de esos cuatro commits, apareciendo en el tradicional ordenamiento por fecha de commit.

Un cambio común para utilizar con el comando `log` en este caso es `--left-right`, el cual muestra en qué lado del rango se encuentra cada commit. Esto ayuda a hacer la información más útil:

```
$ git log --left-right master...experiment
< F
< E
> D
> C
```

Con estas herramientas, se puede hacer saber más facilmente a Git qué commit o commits desea inspecionar.

Organización interactiva

El git viene con unos cuantos scripts que hace que algunas líneas de comando sean más fáciles de usar. Aquí, Veras unos cuantos comandos interactivos que te ayudaran a preparar tus confirmaciones para incluir solo ciertas combinaciones y partes de los archivos. Estás herramientas serán muy útiles si modificas unos cunatos archivos y decides que quieres esos cambios estén en varias confirmaciones enfocadas más que en una gran problemática confirmación. De esta manera, puede asegurarse de que sus confirmaciones sean conjuntos de cambios lógicamente separados y que puedan ser revisados fácilmente por los desarrolladores que trabajan con usted. Si empiezas `git add` con el `-i` o la opción `--interactive` , Git entra en un modo de celda interactiva, mostrando algo como esto:

```
$ git add -i
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:    unchanged        +1/-1 index.html
  3:    unchanged        +5/-1 lib/simplegit.rb

*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now>
```

Puedes ver que este comando te muestra una muy diferente vista de tu área de ensayo – básicamente la misma informacion que con el with `git status` pero un poco más sucinto e

informativo. Muestra los cambios que haz realizado en la izquierda y cambios que no haz hecho a la derecha.

Después de esto viene una sección de comandos. Aquí puedes hacer un sin número de cosas, incluidos los archivos organizados, archivos sin organizar, partes de archivos organizados, agregar archivos sin seguimiento y ver las diferencias de lo que se ha modificado.

Organizar y desorganziar archivos

Si tecleas `2` o `u` en el `What now>` rápidamente, la secuencia de comandos le solicita los archivos que desea representar:

```
What now> 2
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:    unchanged        +1/-1 index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Update>>
```

Para organizar los archivos de TODO and index.html , puedes teclear los números:

```
Update>> 1,2
           staged     unstaged path
* 1:    unchanged        +0/-1 TODO
* 2:    unchanged        +1/-1 index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Update>>
```

El `*` después de cada archivo significa que el archivo fue seleccionado para ser organizado. Si presionas Enter después de no escribir nada en el `Update>>` rapidamente, Git toma cualquier cosa seleccionada y la organiza por ti:

```
Update>>
updated 2 paths

*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 1
           staged     unstaged path
  1:        +0/-1      nothing TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
```

Ahora puedes ver que los archivos de TODO and index.html han sido organizados y el archivo simplegit.rb aun está sin organizar. If you want to unstage the TODO file at this point, you use the `3` or `r` (for revert) option:

```
*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 3
           staged     unstaged path
  1:        +0/-1      nothing TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Revert>> 1
           staged     unstaged path
* 1:        +0/-1      nothing TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Revert>> [enter]
reverted one path
```

Mirando el estatus de tu Git de nuevo, puedes ver que haz desordenado el archivo de TODO:

```
*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 1
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
```

Para ver la diferencia de lo que ya haz ordenado, puedes usar el `6` o `d` (para diferente) comando. Este te muestra una lista de tus archivos organziados, y puedes seleccionar aquellos que quisieras ver la diferencia de su organización. Esto es como especificar el `git diff --cached` en la línea de comando:

```
*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 6
           staged     unstaged path
  1:        +1/-1      nothing index.html
Review diff>> 1
diff --git a/index.html b/index.html
index 4d07108..4335f49 100644
--- a/index.html
+++ b/index.html
@@ -16,7 +16,7 @@ Date Finder

 <p id="out">...</p>

-<div id="footer">contact : support@github.com</div>
+<div id="footer">contact : email.support@github.com</div>

 <script type="text/javascript">
```

Con estos comandos básicos,puedes usar el modo de añadir interactivo para tratar con su área de organización un poco más fácilmente.

Parches de organización

De igual manera es posible para el Git el organizar ciertas partes de archivos y no todos los demás. Por ejemplo, Si haces dos simples cambios en tu archivo simplegit.rb y quieres organizar uno pero no el otro, hacer esto es muy fácil en Git. Desde el prompt interactivo, teclea `5` o `p` (para parche). Git te preguntará que archivos quieres organizar parcialmente; entonces, para cada sección de los archivos seleccionados, mostrará bloques del archivo diferencial y te preguntará si quisierás organizarlos, uno por uno:

```
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index dd5ecc4..57399e0 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -22,7 +22,7 @@ class SimpleGit
   end

   def log(treeish = 'master')
-    command("git log -n 25 #{treeish}")
+    command("git log -n 30 #{treeish}")
   end

   def blame(path)
Stage this hunk [y,n,a,d,/,j,J,g,e,?]?
```

Tienes muchas opciones en este punto. Teclear `?` te mostrará una lista de lo que puedes hacer:

```
Stage this hunk [y,n,a,d,/,j,J,g,e,?]? ?
y - stage this hunk
n - do not stage this hunk
a - stage this and all the remaining hunks in the file
d - do not stage this hunk nor any of the remaining hunks in the file
g - select a hunk to go to
/ - search for a hunk matching the given regex
j - leave this hunk undecided, see next undecided hunk
J - leave this hunk undecided, see next hunk
k - leave this hunk undecided, see previous undecided hunk
K - leave this hunk undecided, see previous hunk
s - split the current hunk into smaller hunks
e - manually edit the current hunk
? - print help
```

Generalmente, teclearías `y` o `n` Si quisieras organizar cada bloque, pero organizar cada uno de ellos en ciertos archivos o saltarte una decisión para algún bloque puede ser de ayuda para más tarde también. Si organizas una parte del archivo y dejas la otra partida sin organizar, su salida de estado se verá así:

```
What now> 1
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:        +1/-1      nothing index.html
  3:        +1/-1        +4/-0 lib/simplegit.rb
```

El estatus del archivo simplegit.rb es interesante. Te muestra que un par de líneas están organizadas y otro par está desorganizado. Has organizado parcialmente este archivo. En este punto, puede salir del script de adición interactivo y ejecutar `git commit` para confirmar los archivos parcialmente organiazdos.

De igual manera no necesitas estar en el modo de adición interactivo para hacer el parcial - organización de archivos – Puedes iniciar el mismo script usando `git add -p` o `git add --patch` en la línea de comando.

Además, puede usar el modo de parche para restablecer parcialmente los archivos con el comando `reset --patch` , para verificar partes de archivos con el comando `checkout --patch` y para esconder partes de archivos con el comando `stash save --patch`. Vamos a entrar en más detalles sobre cada uno de estos a medida que accedemos a usos más avanzados de estos comandos.

Guardado rápido y Limpieza

Muchas veces, cuando has estado trabajando en una parte de tu proyecto, las cosas se encuentran desordenadas y quieres cambiar de ramas por un momento para trabajar en algo más. El problema es que no quieres hacer un commit de un trabajo que va por la mitad, así puedes volver a ese punto más tarde. La respuesta a ese problema es el comando `git stash`.

El guardado rápido toma el desorden de tu directorio de trabajo – que es, tus archivos controlados por la versión modificados y cambios almacenados – y lo guarda en un saco de cambios sin terminar que puedes volver a usar en cualquier momento.

Guardando rápido tu trabajo

Para demostrar, irás a tu proyecto y empezarás a trabajar en un par de archivos y posiblemente en tu etapa uno de cambios. Si ejecutas `git status`, puedes ver tu estado sucio:

```
$ git status
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   index.html

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   lib/simplegit.rb
```

Ahora quieres cambiar de ramas, pero no quieres hacer commit a lo que has estado trabajando todavía; así que le harás un guardado rápido a los cambios. Para poner un nuevo guardado rápido en tus archivos, ejecuta `git stash` o `git stash save`:

```
$ git stash
Saved working directory and index state \
  "WIP on master: 049d078 added the index file"
HEAD is now at 049d078 added the index file
(To restore them type "git stash apply")
```

Tu directorio de trabajo está limpio:

```
$ git status
# On branch master
nothing to commit, working directory clean
```

En este punto, puedes fácilmente cambiar de ramas y hacer trabajos en otro lugar; tus cambios están guardados en tus archivos. Para ver qué guardados rápidos has almacenado, puedes usar `git stash list`:

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
```

In this case, two stashes were done previously, so you have access to three different stashed works. You can reapply the one you just stashed by using the command shown in the help output of the original stash command: `git stash apply`. Si quieres hacer entrada a uno de los guardados rápidos anteriores, puedes especificarlo poniendo su nombre de esta manera: `git stash apply stash@{2}`. Si no especificas un guardado, el Git adopta el guardado más reciente e intenta hacerle entrada:

```
$ git stash apply
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#      modified:   index.html
#      modified:   lib/simplegit.rb
#
```

Puedes ver que el Git remodifica los archivos que revertiste cuando hiciste el guardado rápido. En este caso, tenías un directorio de trabajo despejado cuando intentaste hacer entrada al guardado, e intentaste hacerle entrada en la misma rama en la que la guardaste; pero tener un directorio de trabajo despejado y usarlo en la misma rama no es necesario para hacerle entrada a un guardado con éxito. Puedes almacenar un guardado en una rama, cambiar a otra rama luego, e intentar volver a hacerle entrada a los cambios. También puedes tener archivos modificados y sin aplicar en tu directorio de trabajo cuando des entrada a un guardado – Git te da conflictos de cambinación si algo ya no se usa de manera limpia.

Los cambios a tus archivos fueron reaplicados, pero el archivo que tu guardaste antes no fue realmacenado. Para hacer eso, tienes que ejecuar el comando `git stash apply` con una opción de `--index` para decirle al comando que intente reaplicar los cambios almacenados. Si en cambio lo hubieras ejecutado, lo habrías vuelto a tener en su posición original:

```
$ git stash apply --index
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#      modified:   index.html
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#      modified:   lib/simplegit.rb
#
```

La opción de hacer entrada sólo intenta hacer entrada al trabajo guardado –lo continúas teniendo en tus archivos. Para removerlo, puedes ejecutar `git stash drop` con el nombre del guardado a eliminar:

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
$ git stash drop stash@{0}
Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)
```

También puedes ejecutar `git stash pop` para hacer entrada al guardado y luego eliminarlo inmediatamente de tus archivos.

Guardado rápido creativo

Hay algunas pocas variantes de guardado rápido que pueden ser útiles también. La primera opción que es muy popular es la opción `--keep-index` para el comando `stash save`. Esto le dice a Git que no guarde nada que tú ya hayas almacenado con el comando `git add`.

Esto puede ser útil de verdad si has hecho un buen número de cambios, pero sólo quiere aplicar permanentemente algunos de ellos y luego regresar al resto de cambios en una siguient ocasión.

```
$ git status -s
M  index.html
 M lib/simplegit.rb

$ git stash --keep-index
Saved working directory and index state WIP on master: 1b65b17 added the index file
HEAD is now at 1b65b17 added the index file

$ git status -s
M  index.html
```

Otra cosa común que puede que quieras hacer con tus guardados es hacer un guardado rápido de los archivos que no están bajo control de la versión al igual que con los que lo están. Por defecto, git stash` sólamente guardará archivos que ya están en el índice. Si especificas `--include-untracked` o `-u`, el Git tambien hará un guardado rápido de cualquier archivo que no esté bajo control de la versión que hayas creado.

```
$ git status -s
M  index.html
 M lib/simplegit.rb
?? new-file.txt

$ git stash -u
Saved working directory and index state WIP on master: 1b65b17 added the index file
HEAD is now at 1b65b17 added the index file

$ git status -s
$
```

Finalmente, si especificas el flag `--patch`, el Git no hará guardado rápido de todo lo que es modificado, pero, en su lugar, te recordará cuales de los cambios te gustaría guardar y cuales te gustaría mantener en tu trabajo directamente.

```
$ git stash --patch
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 66d332e..8bb5674 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -16,6 +16,10 @@ class SimpleGit
         return `#{git_cmd} 2>&1`.chomp
       end
     end
+
+    def show(treeish = 'master')
+      command("git show #{treeish}")
+    end

 end
 test
Stash this hunk [y,n,q,a,d,/,e,?]? y

Saved working directory and index state WIP on master: 1b65b17 added the index file
```

Creando una Rama desde un Guardado Rápido

Si haces guardado rápido de algo de trabajo, déjalo ahí por un rato, y continúa en la rama de la cual hiciste guardado rápido de tu trabajo, puede que tengas problemas rehaciendo entrada al trabajo. Si la entrada intenta modificar un archivo que desde entonces has modificado, tendrás un conflicto de combinación y tendrás que intentar resolverlo. Si quieres una forma más fácil de probar los cambios guardados de nuevo, puedes ejecutar `git stash branch`, el cual crea una nueva rama para ti, verifica el commit en el que estabas cuando hiciste guardado rápido de tu trabajo, recrea tu trabajo allí, y luego arroja el guardado rápido si la entrada se realiza con éxito:

```
$ git stash branch testchanges
Switched to a new branch "testchanges"
# On branch testchanges
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#      modified:   index.html
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#      modified:   lib/simplegit.rb
#
Dropped refs/stash@{0} (f0dfc4d5dc332d1cee34a634182e168c4efc3359)
```

Este es un buen método rápido para recuperar rápidamente trabajos guardados y trabajar en una nueva rama.

Limpiando tu Directorio de Trabajo

Finalmente, puede que no quieras hacer guardado rápido de algo de trabajo o de archivos en tu directorio de trabajo, pero quieres desaserte de ellos. El comando `git clean` hará esto por ti.

Algunas razones comunes para esto pueden ser para remover archivos cruft que han sido generados por herramientas de combinación o externas o para eliminar viejos archivos de versión con el fin de ejecutar una versión limpia.

Querrás ser más bien delicado con este comando, ya que está diseñado para eliminar archivos de tu directorio de trabajo que no están siendo tomados en cuenta. Si cambias de opinión, muchas veces no hay restauración para el contenido de esos archivos. Una opción más segura es ejecutar `git stash --all` para eliminar todo, pero lo almacena en un guardado.

Asumiendo que quieres eliminar los archivos cruft o limpiar tu directorio de trabajo, puedes hacerlo con `git clean`. Para remover los archivos que no están bajo el control de la versión en tu directorio de trabajo, puedes ejecutar `git clean -f -d`, el cual remueve cualquier archivo y también cualquier subdirectorio que se vuelva vacío como resultado. El `-f` significa *fuerza* o “realmente hace esto”.

Si alguna vez quieres ver que haría, puedes ejecutar el comando con la opción `-n` que significa ‘`haz un arranque en seco y dime que habías eliminado ''.

```
$ git clean -d -n
Would remove test.o
Would remove tmp/
```

Por defecto, el comando `git clean` solo removerá archivos que no sean controlados que no sean ignorados. Cualquier archivo que empareje en patrón en tu `.gitignore` u otros archivos ignorados no serán removidos. Si quieres eliminar esos archivos también, como eliminar todos `.o` files

generados por la versión, así puedes hacer una versión completamente limpia, puedes añadir un `-x` al comando en limpio.

```
$ git status -s
 M lib/simplegit.rb
?? build.TMP
?? tmp/

$ git clean -n -d
Would remove build.TMP
Would remove tmp/

$ git clean -n -d -x
Would remove build.TMP
Would remove test.o
Would remove tmp/
```

Si no sabes lo que el comando `git clean` va a hacer, siempre ejecútalo con un `-n` primero para estar seguro antes de cambiar el `-n` a -f`y hacerlo de verdad. La otra forma en la que puedes ser cuidadoso con el proceso es ejecutarlo con el `-i o con la flag “interactive”.

Esto ejecutará el comando en limpio en un modo interactivo.

```
$ git clean -x -i
Would remove the following items:
  build.TMP  test.o
*** Commands ***
    1: clean                2: filter by pattern    3: select by numbers    4: ask
each             5: quit
    6: help
What now>
```

De esta forma puedes decidir por cada archivo individualmente o especificar los terminos para la eliminación de forma interactiva.

Firmando tu trabajo

Git es criptográficamente seguro, pero no es a prueba de tontos. Si estás tomando trabajo de otros de internet y quieres verificar que los commits son realmente de fuentes seguras, Git tiene unas cuantas maneras de firmar y verificar utilizando GPG.

Introducción a GPG

Antes que nada, si quieres firmar cualquier cosa necesitas tener configurado GPG y tu llave personal instalada.

```
$ gpg --list-keys
/Users/schacon/.gnupg/pubring.gpg
---------------------------------
pub   2048R/0A46826A 2014-06-04
uid                  Scott Chacon (Git signing key) <schacon@gmail.com>
sub   2048R/874529A9 2014-06-04
```

Si no tienes una llave instalada, puedes generar una con `gpg --gen-key`.

```
gpg --gen-key
```

Una vez que tengas una llave privada para firmar, puedes configurar Git para usarla y firmar cosas configurando la opción `user.signingkey`.

```
git config --global user.signingkey 0A46826A
```

Ahora Git usará tu llave por defecto para firmar tags y commits si tu quieres.

Firmando Tags

Si tienes una llave GPG privada configurada, ahora puedes usarla para firmar tags. Todo lo que tienes que hacer es usar `-s` en lugar de `-a`:

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'

You need a passphrase to unlock the secret key for
user: "Ben Straub <ben@straub.cc>"
2048-bit RSA key, ID 800430EB, created 2014-05-04
```

Si ejecutas `git show` en ese tag, puedes ver tu firma GPG adjunta a él:

```
$ git show v1.5
tag v1.5
Tagger: Ben Straub <ben@straub.cc>
Date:   Sat May 3 20:29:41 2014 -0700

my signed 1.5 tag
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1

iQEcBAABAgAGBQJTZbQlAAoJEF0+sviABDDrZbQH/09PfE51KPVPlanr6q1v4/Ut
LQxfojUWiLQdg2ESJItkcuweYg+kc3HCyFejeDIBw9dpXt00rY26p05qrpnG+85b
hM1/PswpPLuBSr+oCIDj5GMC2r2iEKsfv2fJbNW8iWAXVLoWZRF8B0MfqX/YTMbm
ecorc4iXzQu7tupRihslbNkfvfciMnSDeSvzCpWAHl7h8Wj6hhqePmLm9lAYqnKp
8S5B/1SSQuEAjRZgI4IexpZoeKGVDptPHxLLS38fozsyi0QyDyzEgJxcJQVMXxVi
RUysgqjcpT8+iQM1PblGfHR4XAhuOqN5Fx06PSaFZhqvWFezJ28/CLyX5q+oIVk=
=EFTF
-----END PGP SIGNATURE-----

commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number
```

Verficando Tags

Para verificar un tag firmado, usa `git tag -v [nombre-de-tag]`. Este comando usa GPG para verificar la firma. Necesitas tener guardada la llave pública del usuario para que esto funcione de manera apropiada:

```
$ git tag -v v1.4.2.1
object 883653babd8ee7ea23e6a5c392bb739348b1eb61
type commit
tag v1.4.2.1
tagger Junio C Hamano <junkio@cox.net> 1158138501 -0700

GIT 1.4.2.1

Minor fixes since 1.4.2, including git-mv and git-http with alternates.
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Good signature from "Junio C Hamano <junkio@cox.net>"
gpg:                 aka "[jpeg image of size 1513]"
Primary key fingerprint: 3565 2A26 2040 E066 C9A7  4A7D C0C6 D9A4 F311 9B9A
```

Si no tienes la llave pública de quien firmó, obtendrás algo como esto en cambio:

```
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Can't check signature: public key not found
error: could not verify the tag 'v1.4.2.1'
```

Firmando Commits

En versiones más recientes de Git (v1.7.9 en adelante), ahora puedes firmar commits individuales. Si estás interesado en firmar commits directamente en lugar de solo los tags, todo lo que necesitas hacer es agregar un `-S` a tu comando `git commit`.

```
$ git commit -a -S -m 'signed commit'

You need a passphrase to unlock the secret key for
user: "Scott Chacon (Git signing key) <schacon@gmail.com>"
2048-bit RSA key, ID 0A46826A, created 2014-06-04

[master 5c3386c] signed commit
 4 files changed, 4 insertions(+), 24 deletions(-)
 rewrite Rakefile (100%)
 create mode 100644 lib/git.rb
```

Para ver y verificar las firmas, también existe una opción `--show-signature` para `git log`.

```
$ git log --show-signature -1
commit 5c3386cf54bba0a33a32da706aa52bc0155503c2
gpg: Signature made Wed Jun  4 19:49:17 2014 PDT using RSA key ID 0A46826A
gpg: Good signature from "Scott Chacon (Git signing key) <schacon@gmail.com>"
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Jun 4 19:49:17 2014 -0700

    signed commit
```

Adicionalmente, puedes configurar `git log` para verificar cualquier firma que encuentre y listarlas en su salida con el formato `%G?`.

```
$ git log --pretty="format:%h %G? %aN  %s"

5c3386c G Scott Chacon  signed commit
ca82a6d N Scott Chacon  changed the version number
085bb3b N Scott Chacon  removed unnecessary test code
a11bef0 N Scott Chacon  first commit
```

Aquí podemos ver que solo el último commit es firmado y válido y los commits previos no.

En Git 1.8.3 y posteriores, "git merge" y "git pull" pueden ser configurados para inspeccionar y rechazar cualquier commit que no adjunte una firma GPG de confianza con el comando `--verify`

`-signatures`.

Si se usa esta opción cuando se fusiona una rama y esta contiene commits que no están firmados y son válidos, la fusión no funcionará.

```
$ git merge --verify-signatures non-verify
fatal: Commit ab06180 does not have a GPG signature.
```

Si una fusión contiene solo commits válidos y firmados, el comando merge mostrará todas las firmas que ha revisado y después procederá con la fusión.

```
$ git merge --verify-signatures signed-branch
Commit 13ad65e has a good GPG signature by Scott Chacon (Git signing key)
<schacon@gmail.com>
Updating 5c3386c..13ad65e
Fast-forward
 README | 2 ++
 1 file changed, 2 insertions(+)
```

También se puede utilizar la opción `-S` junto con el mismo comando `git merge` para firmar el commit resultante. El siguiente ejemplo verifica que cada commit en la rama por ser fusionada esté firmado y también firma el commit resultado de la fusión.

```
$ git merge --verify-signatures -S  signed-branch
Commit 13ad65e has a good GPG signature by Scott Chacon (Git signing key)
<schacon@gmail.com>

You need a passphrase to unlock the secret key for
user: "Scott Chacon (Git signing key) <schacon@gmail.com>"
2048-bit RSA key, ID 0A46826A, created 2014-06-04

Merge made by the 'recursive' strategy.
 README | 2 ++
 1 file changed, 2 insertions(+)
```

Todos deben firmar

Firmar tags y commits es grandioso, pero si decides usar esto en tu flujo de trabajo normal, tendrás que asegurar que todos en el equipo entiendan cómo hacerlo. Si no, terminarás gastando mucho tiempo ayudando a las personas a descubrir cómo reescribir sus commits con versiones firmadas. Asegúrate de entender GPG y los beneficios de firmar cosas antes de adoptarlo como parte de tu flujo de trabajo normal.

Buscando

Con casi cualquier tamaño de código de base, a menudo necesitará encontrar en dónde se llama o

define una función, o encontrar el historial de un método. Git proporciona un par de herramientas útiles para examinar el código y hacer commit a las almacenadas en su base de datos de forma rápida y fácil. Vamos a revisar algunos de ellos.

Git Grep

Git se envía con un comando llamado `grep` que le permite buscar fácilmente a través de cualquier árbol o directorio de trabajo con commit por una cadena o expresión regular. Para estos ejemplos, veremos el código fuente de Git.

Por defecto, mirará a través de los archivos en su directorio de trabajo. Puede pasar `-n` para imprimir los números de línea donde Git ha encontrado coincidencias.

```
$ git grep -n gmtime_r
compat/gmtime.c:3:#undef gmtime_r
compat/gmtime.c:8:      return git_gmtime_r(timep, &result);
compat/gmtime.c:11:struct tm *git_gmtime_r(const time_t *timep, struct tm *result)
compat/gmtime.c:16:     ret = gmtime_r(timep, result);
compat/mingw.c:606:struct tm *gmtime_r(const time_t *timep, struct tm *result)
compat/mingw.h:162:struct tm *gmtime_r(const time_t *timep, struct tm *result);
date.c:429:             if (gmtime_r(&now, &now_tm))
date.c:492:             if (gmtime_r(&time, tm)) {
git-compat-util.h:721:struct tm *git_gmtime_r(const time_t *, struct tm *);
git-compat-util.h:723:#define gmtime_r git_gmtime_r
```

Hay una serie de opciones interesantes que puede proporcionar el comando `grep`.

Por ejemplo, en lugar de la llamada anterior, puede hacer que Git resuma el resultado simplemente mostrándole qué archivos coinciden y cuántas coincidencias hay en cada archivo con la opción `--count`:

```
$ git grep --count gmtime_r
compat/gmtime.c:4
compat/mingw.c:1
compat/mingw.h:1
date.c:2
git-compat-util.h:2
```

Si quiere ver qué método o función piensa que ha encontrado una coincidencia, puede pasar `-p`:

```
$ git grep -p gmtime_r *.c
date.c=static int match_multi_number(unsigned long num, char c, const char *date, char
*end, struct tm *tm)
date.c:         if (gmtime_r(&now, &now_tm))
date.c=static int match_digit(const char *date, struct tm *tm, int *offset, int
*tm_gmt)
date.c:         if (gmtime_r(&time, tm)) {
```

Así que aquí podemos ver que se llama a `gmtime_r` en las funciones `match_multi_number` y `match_digit` en el archivo date.c.

También puede buscar combinaciones complejas de cadenas con el indicador `--and`, que asegura que múltiples coincidencias estén en la misma línea. Por ejemplo, busquemos cualquier línea que defina una constante con las cadenas "LINK" o "BUF_MAX" en ellas en la base del código de Git en una versión 1.8.0 anterior.

Aquí también usaremos las opciones `--break` y `--heading` que ayudan a dividir el resultado en un formato más legible.

```
$ git grep --break --heading \
    -n -e '#define' --and \( -e LINK -e BUF_MAX \) v1.8.0
v1.8.0:builtin/index-pack.c
62:#define FLAG_LINK (1u<<20)

v1.8.0:cache.h
73:#define S_IFGITLINK  0160000
74:#define S_ISGITLINK(m)       (((m) & S_IFMT) == S_IFGITLINK)

v1.8.0:environment.c
54:#define OBJECT_CREATION_MODE OBJECT_CREATION_USES_HARDLINKS

v1.8.0:strbuf.c
326:#define STRBUF_MAXLINK (2*PATH_MAX)

v1.8.0:symlinks.c
53:#define FL_SYMLINK  (1 << 2)

v1.8.0:zlib.c
30:/* #define ZLIB_BUF_MAX ((uInt)-1) */
31:#define ZLIB_BUF_MAX ((uInt) 1024 * 1024 * 1024) /* 1GB */
```

El comando `git grep` tiene algunas ventajas sobre los comandos de búsqueda normales como `grep` y `ack`. La primera es que es realmente rápido, la segunda es que puedes buscar a través de cualquier árbol en Git, no solo en el directorio de trabajo. Como vimos en el ejemplo anterior, buscamos términos en una versión anterior del código fuente de Git, no en la versión que estaba actualmente verificada.

Búsqueda de Registro de Git

Quizás no está buscando **dónde** existe un término, sino **cuando** existió o se introdujo. El comando `git log` tiene varias herramientas potentes para encontrar commits específicas por el contenido de sus mensajes o incluso el contenido de las diferencias que introducen.

Si queremos saber, por ejemplo, cuándo se introdujo originalmente la constante `ZLIB_BUF_MAX`, podemos decirle ac Git que solo nos muestre los commits que agregaron o eliminaron esa cadena con la opción `-S`.

```
$ git log -SZLIB_BUF_MAX --oneline
e01503b zlib: allow feeding more than 4GB in one go
ef49a7a zlib: zlib can only process 4GB at a time
```

Si miramos la diferencia de esos commits, podemos ver que en `ef49a7a` se introdujo la constante y en `e01503b` se modificó.

Si necesita ser más específico, puede proporcionar una expresión regular para buscar con la opción `-G`.

Búsqueda de Registro de Línea

Otra búsqueda de registro bastante avanzada que es increíblemente útil es la búsqueda del historial de línea. Esta es una adición bastante reciente y no muy conocida, pero puede ser realmente útil. Se llama con la opción `-L` a `git log` y le mostrará el historial de una función o línea de código en su base de código.

Por ejemplo, si quisiéramos ver cada cambio realizado en la función `git_deflate_bound` en el archivo `zlib.c`, podríamos ejecutar `git log -L :git_deflate_bound:zlib.c`. Esto intentará descubrir cuáles son los límites de esa función y luego examinará el historial y nos mostrará cada cambio que se hizo a la función como una serie de parches de nuevo cuando se creó la función por primera vez.

```
$ git log -L :git_deflate_bound:zlib.c
commit ef49a7a0126d64359c974b4b3b71d7ad42ee3bca
Author: Junio C Hamano <gitster@pobox.com>
Date:   Fri Jun 10 11:52:15 2011 -0700

    zlib: zlib can only process 4GB at a time

diff --git a/zlib.c b/zlib.c
--- a/zlib.c
+++ b/zlib.c
@@ -85,5 +130,5 @@
-unsigned long git_deflate_bound(z_streamp strm, unsigned long size)
+unsigned long git_deflate_bound(git_zstream *strm, unsigned long size)
 {
-       return deflateBound(strm, size);
+       return deflateBound(&strm->z, size);
 }


commit 225a6f1068f71723a910e8565db4e252b3ca21fa
Author: Junio C Hamano <gitster@pobox.com>
Date:   Fri Jun 10 11:18:17 2011 -0700

    zlib: wrap deflateBound() too

diff --git a/zlib.c b/zlib.c
--- a/zlib.c
+++ b/zlib.c
@@ -81,0 +85,5 @@
+unsigned long git_deflate_bound(z_streamp strm, unsigned long size)
+{
+       return deflateBound(strm, size);
+}
+
```

Si Git no puede encontrar la forma de relacionar una función o método en su lenguaje de programación, también puede proporcionarle una expresión regular. Por ejemplo, esto habría hecho lo mismo: `git log -L '/unsigned long git_deflate_bound/',/^}/:zlib.c`. También podría darle un rango de líneas o un único número de línea y obtendrá el mismo tipo de salida.

Reescribiendo la Historia

Muchas veces, al trabajar con Git, vas a querer confirmar tu historia por alguna razón. Una de las grandes cualidades de Git es que te permite tomar decisiones en el último momento. Puede decidir qué archivos entran en juego antes de comprometerse con el área de ensayo, puedes decidir que no querías estar trabajando en algo todavía con el comando de alijos, y puedes reescribir confirmaciones que ya hayan pasado haciendo parecer que fueron hechos de diferente manera. Esto puede desenvolverse en el cambio de las confirmaciones, cambiando mensajes o modificando

los archivos en un cometido, aplastando o dividiendo confirmaciones enteramente – todo antes de que compartas tu trabajo con otras personas.

En esta sección, verás cómo complementar esas tareas tan útiles que harán parece que la confirmación de tu historia parezca del modo en el cual quisiste compartirla.

Cambiando la última confirmación

Cambiar la última confirmación es probablemente lo más común que le harás a tu historia. Comúnmente querrás hacer dos cosas en tu ultima confirmación: cambiar la confirmación del mensaje, o cambiar la parte instantánea que acabas de agregar sumando, cambiando y removiendo archivos.

Si solamente quieres cambiar la confirmación del mensaje final, es muy sencillo:

```
$ git commit --amend
```

Esto te envía al editor de texto, el cual tiene tu confirmación final, listo para modificarse en el mensaje. Cuando guardes y cierres el editor, el editor escribe una nueva confirmación conteniendo el mensaje y lo con tu ultima confirmación.

Si ya ha cambiado su última confirmación y luego quiere cambiar la instantánea que confirmo al agregar o cambiar archivos, esto es por que posiblemente olvidó agregar un archivo recién creado cuando se confirmó originalmente, el proceso trabaja prácticamente de la misma manera. Tu manejas los cambios que quieras editando el archivo y oprimiendo `git add` en este o `git rm` a un archivo adjunto, y el subsecuente git commit --amend toma tu área de trabajo actual y la vuelve una instantánea para la nueva confirmación.

Debes ser cuidadoso con esta técnica porque puedes modificar los cambios del SHA-1 de la confirmación Es como un muy pequeño rebase – no necesitas modificar tu ultima confirmación si ya el haz puesto.

Cambiando la confirmación de múltiples mensajes

Para modificar una confirmación que está más atrás en tu historia, deberás aplicar herramientas más complejas Git no tiene una herramienta para modificar la historia, pero puedes usar la herramienta de rebase para rebasar ciertas series confirmaciones en el HEAD en el que se basaron originalmente en lugar de moverlos a otro. Con la herramienta interactiva del rebase, puedes parar justo después de cada confirmación que quieras modificar y cambiar su mensaje, añadir archivos, o hacer cualquier cosa que quieras Puedes ejecutar el rebase interactivamente agregando el `-i` option to `git rebase`. De igual manera debes indicar que tan atrás quieres regresar para reescribir las confirmaciones escribiendo en el comando cual confirmación quieres rebasar.

Por ejemplo, si quieres cambiar las confirmaciones de los tres últimos mensajes, o cualquiera de los mensajes de confirmación de ese grupo, proporcionas un argumento para lo `git rebase -i` que quieras modificar de tu ultima confirmación, el cual es `HEAD~2^` or `HEAD~3` . Debería ser más fácil el recordar el ~3 porque estás tratando de editar las últimas tres confirmaciones; pero ten en mente que estás designando actualmente cuatro confirmaciones atrás, la parte del ultimo cometido que

deseas editar:

```
$ git rebase -i HEAD~3
```

Recuerda de Nuevo es este es un comando de rebase – cualquier confirmación incluida en el rango de `HEAD~3..HEAD` será reescrita, aun si cambias el mensaje o no. No incluyas cualquier confirmación que ya hayas enviado al servidor central – si lo haces esto confundirá a los demás desarrolladores proporcionando una versión alternativa del mismo cambio.

Utilizar este comando te da una lista de las confirmaciones en tu editor de texto que se ve como este:

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

Es importante el notar que estas confirmaciones son escuchadas en el orden contrario de la que tú normalmente las verías usando el comando de `log`. Si utilizaras un comando de `log`, verías algo como esto.

```
$ git log --pretty=format:"%h %s" HEAD~3..HEAD
a5f4a0d added cat-file
310154e updated README formatting and added blame
f7f3f6d changed my name a bit
```

Nótese que el orden esta al revés. El rebase interactivo te da un script que va a utilizarse. Este empezará en la confirmación que especificas en la línea de comandos (`HEAD~3`) y reproducir los cambios introducidos en cada una de estas confirmaciones de arriba a abajo. Este acomoda los más viejos en la parte de arriba, y va bajando hasta los más nuevos, porque ese será el primero en

reproducirse

Necesitaras editar el script para que se detenga en la confirmación que quieres editar. Para hacer eso, cambia la palabra `pick` por la frase `edit` para cada una de las confirmaciones en las que quieres que el script se detenga. Por ejemplo, para modificar solamente la tercera confirmación del mensaje, cambiarias el archivo para que se viera algo así:

```
edit f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

Cuando guardes y salgas del editor, Git te enviará atrás a la última confirmación en la lisa y te llevará a la línea de comando con el siguiente mensaje:

```
$ git rebase -i HEAD~3
Stopped at f7f3f6d... changed my name a bit
You can amend the commit now, with

       git commit --amend

Once you’re satisfied with your changes, run

       git rebase --continue
```

Estas instrucciones te dirán exactamente qué hacer. Type

```
$ git commit --amend
```

Cambia la confirmación del mensaje, y sal del editor. Then, run

```
$ git rebase --continue
```

Este comando te permitirá aplicar las otras dos confirmaciones automáticamente, y después de esto estás listo. Si decides cambiar y elegir editar en más líneas, puedes repetir estos pasos para cada confirmación que cambies en cada edición. Cada vez, Git se parará, permitiéndote modificar la confirmación y continuar cuando hayas terminado

Reordenando Confirmaciones

De igual manera puedes usar rebases interactivos para reordenar o remover confirmaciones enteramente. Si quieres remover la “added cat-file” confirmación y cambiar el orden en el cual las otras dos confirmaciones son introducidas, puedes cambiar el rebase en el script de esto:

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

A esto:

```
pick 310154e updated README formatting and added blame
pick f7f3f6d changed my name a bit
```

Cuando guardes y salgas del editor, Git te recordará de tu rama de padres de estas confirmaciones, aplicando 310154e y después f7f3f6d, y después se parará. Cambias efectivamente el orden de esas confirmaciones y eliminas la "added cat-file" confirmación completamente.

Unir confirmaciones

También es posible el tomar series de confirmaciones y unirlas todas en una sola confirmación con la herramienta interactiva de rebase. El script pone instrucciones que en el mensaje de rebase:

```
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

Si, en vez de "`pick" o`"edit", especificas "squash", Git aplica a ambos este cambio y los cambia directamente después y hace que las confirmaciones se unan. Entonces, si quieres convertir en una única confirmación de estas tres confirmaciones, deberás hacer que el script se vea como esto:

```
pick f7f3f6d changed my name a bit
squash 310154e updated README formatting and added blame
squash a5f4a0d added cat-file
```

Cuando guardes te salgas del editor, Git aplicará a los tres el cambio y después te dirigirá en el editor para fusionar los tres mensajes de la confirmación:

```
# This is a combination of 3 commits.
# The first commit's message is:
changed my name a bit

# This is the 2nd commit message:

updated README formatting and added blame

# This is the 3rd commit message:

added cat-file
```

Cuando guardes eso, tendrás una única confirmación que introducirá los cambios de las tres previas confirmaciones.

Dividiendo una confirmación

Dividir una confirmación la deshace y después realiza etapas parcialmente de las confirmaciones tantas veces como confirmaciones desees finalizar. Por ejemplo, suponiendo que quieres dividir la confirmación de en medio de tus tres confirmaciones. En vez de "`updated README formatting and added blame", quieres dividirla en dos confirmaciones: "updated README formatting" para la primera, y "added blame" para la segunda. Puedes hacer eso en el `rebase -i` script cambiando la instruccion en la confirmacion que quieres dividir a "edit":

```
pick f7f3f6d changed my name a bit
edit 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

¿Entonces, cuando el script te envié a la línea de comandos, tu reseteas esa confirmación, tomas los cambios que se han hecho, y creas múltiples confirmación fuera de ellas? Cuando guardes y salgas del editor, Git te enviará al padre de la primer confirmación en tu lista, Aplica a la primera confirmación (`f7f3f6d`), aplicando a la segunda (`310154e`) y te enviará directamente a la consola. Ahí, puedes hacer un reseteo mixto de esa confirmación con el `git reset HEAD^`, el que efectivamente deshace las confirmaciones en los archivos hasta que tengas. Ahora puedes organizar y confirmar los archivos hasta que tenga varias confirmaciones y ejecutar `git rebase --continuar` cuando haya terminado:

```
$ git reset HEAD^
$ git add README
$ git commit -m 'updated README formatting'
$ git add lib/simplegit.rb
$ git commit -m 'added blame'
$ git rebase --continue
```

Git aplica la última confirmación (`a5f4a0d`) en el script, y tu historia quedaría de esta manera:

```
$ git log -4 --pretty=format:"%h %s"
1c002dd added cat-file
9b29157 added blame
35cfb2b updated README formatting
f3cc40e changed my name a bit
```

Una vez de Nuevo, esto cambia el SHA-1s de todas tus confirmaciones en tu lista, así que asegúrate de que ninguna confirmación esté en esa lista que ya has puesto en un repositorio compartido.

La opción nuclear: filtrar-ramificar

Existe otra opción en la parte de volver a escribir la historia que puedes usar si necesitas reescribir un gran número de confirmaciones de una manera que se puedan scriptear – de hecho, cambiar tu dirección de e-mail o remover cualquier archivo en las confirmaciones. El comando es `filter-branch`, y este puede reescribir una gran cantidad de franjas de su historia, así que probablemente no lo deberías usar a menos que tu proyecto aún no sea público y otra persona no se haya basado en sus confirmaciones que estás a punto de reescribir. Como sea, podría ser muy útil. Aprenderás unas cuantas maneras muy comunes de obtener una idea de algunas de las cosas que es capaz de hacer.

Remover un archivo de cada confirmación

Esto ocurre comúnmente. Esto ocurre comúnmente. Alguien accidentalmente confirma un gran numero binario de un archivo con un irreflexivo `git add .`, y quieres removerlo de todas partes. Suponiendo que accidentalmente confirmaste un archivo que contenía contraseña y quieres volverlo un proyecto abierto. `filter-branch` es la herramienta que tu probablemente quieres usar para limpiar toda tu historia. Para remover un archivo nombrado passwords.txt de tu historia complete puedes usar el `--tree-filter` option to `filter-branch`:

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
Ref 'refs/heads/master' was rewritten
```

El `--tree-filter` inicia el comando especifico después de cada revisión del proyecto y este entonces vuelve a confirmar los resultados. En este caso, deberías remover el archivo llamado passwords.txt de cada instantánea, aun si existe o no. si quieres remover todas las confirmaciones accidentales del respaldo del editor de archivos, puedes iniciar algo como el `git filter-branch --tree-filter 'rm -f *~' HEAD`.

Deberías ser capaz de ver la reescripción de confirmaciones y estructuras del Git y luego debes mover el puntero de la rama al final. Es generalmente una buena idea hacer esto en una parte de prueba de la rama y después hacer un hard-reset de tu rama principal después de haber determinado que el resultado es lo que realmente deseas. Para iniciar `filter-branch` en todas las ramas, puedes poner `--all` en el comando.

Hacer que un subdirectorio sea la nueva raíz

Suponiendo que has hecho un importe de otro centro de Sistema de control y tienes subdirecciones que no tienen ningún sentido (tronco, etiquetas, etc). . Si quieres hacer que el `tronco` subdirectorio sea el nuevo proyecto de la raíz de cada confirmación, `filter-branch`te puede ayudar a hacer eso, también:

```
$ git filter-branch --subdirectory-filter trunk HEAD
Rewrite 856f0bf61e41a27326cdae8f09fe708d679f596f (12/12)
Ref 'refs/heads/master' was rewritten
```

Ahora la raíz de tu Nuevo proyecto es la que solía estar en el `tronco` subdirectorio cada vez. Git automáticamente remueve confirmaciones que no afecten al subdirectorio.

Cambiar la dirección de e-mail globalmente

Otro caso comun es que olvides iniciar el git config para poner tu nombre tu dirección de e-mail antes de que hayas empezado a trabajar, o tal vez quieres abrir un proyecto en el trabajo y cambiar todo tu e-mail de trabajo por tu e-mail personal. De cualquier caso, puedes cambiar la dirección de e-mail en multiples confirmaciones en un lote con `filter-branch` igual. Necesitas ser cuidadoso de cambiar solo las direcciones de e-mail que son tuyas, de manera que debes usar `--commit-filter`:

```
$ git filter-branch --commit-filter '
        if [ "$GIT_AUTHOR_EMAIL" = "schacon@localhost" ];
        then
                GIT_AUTHOR_NAME="Scott Chacon";
                GIT_AUTHOR_EMAIL="schacon@example.com";
                git commit-tree "$@";
        else
                git commit-tree "$@";
        fi' HEAD
```

Esto va a traves de la reescripcion de cada confirmacion para tener tu nuva direccion. Por que cada confirmacion contiene el SHA-1 de sus padres, este comando cambia cada confirmacion del SHA-1 en tu historia, no solamente aquellos en los cuales el e-mail es el mismo o encaja.

Reiniciar Desmitificado

Antes de pasar a herramientas más especializadas, hablemos de `reset` y `checkout`. Estos comandos son dos de las partes más confusas de Git cuando los encuentras por primera vez. Hacen tantas cosas que parece imposible comprenderlas realmente y emplearlas adecuadamente. Para esto, recomendamos una metáfora simple.

Los Tres Árboles

Una manera más fácil de pensar sobre `reset` y `checkout` es a través del marco mental de Git como administrador de contenido de tres árboles diferentes. Por “árbol” aquí realmente queremos decir

“colección de archivos”, no específicamente la estructura de datos. (Hay algunos casos donde el índice no funciona exactamente como un árbol, pero para nuestros propósitos es más fácil pensarlo de esta manera por ahora).

Git como sistema maneja y manipula tres árboles en su operación normal:

Árbol	Rol
HEAD	Última instantánea del commit, próximo padre
Índice	Siguiente instantánea del commit propuesta
Directorio de Trabajo	Caja de Arena

El HEAD

HEAD es el puntero a la referencia de bifurcación actual, que es, a su vez, un puntero al último commit realizado en esa rama. Eso significa que HEAD será el padre del próximo commit que se cree. En general, es más simple pensar en HEAD como la instantánea de **su última commit**.

De hecho, es bastante fácil ver cómo se ve esa instantánea. Aquí hay un ejemplo de cómo obtener la lista del directorio real y las sumas de comprobación SHA-1 para cada archivo en la instantánea de HEAD:

```
$ git cat-file -p HEAD
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
author Scott Chacon  1301511835 -0700
committer Scott Chacon  1301511835 -0700

commit inicial

$ git ls-tree -r HEAD
100644 blob a906cb2a4a904a152...   README
100644 blob 8f94139338f9404f2...   Rakefile
040000 tree 99f1a6d12cb4b6f19...   lib
```

Los comandos `cat-file` y `ls-tree` son comandos de “fontanería” que se usan para cosas de nivel inferior y que no se usan realmente en el trabajo diario, pero nos ayudan a ver qué está sucediendo aquí.

El Índice

El índice es su **siguiente commit propuesta**. También nos hemos estado refiriendo a este concepto como el “Área de Preparación” de Git ya que esto es lo que Git ve cuando ejecutas `git commit`.

Git rellena este índice con una lista de todos los contenidos del archivo que fueron revisados por última vez en su directorio de trabajo y cómo se veían cuando fueron revisados originalmente. A continuación, reemplace algunos de esos archivos con nuevas versiones de ellos, y `git commit` los convierte en el árbol para una nueva commit.

```
$ git ls-files -s
100644 a906cb2a4a904a152e80877d4088654daad0c859 0	README
100644 8f94139338f9404f26296befa88755fc2598c289 0	Rakefile
100644 47c6340d6459e05787f644c2447d2595f5d3a54b 0	lib/simplegit.rb
```

Nuevamente, aquí estamos usando `ls-files`, que es más un comando entre bastidores que le muestra a qué se parece actualmente su índice.

El índice no es técnicamente una estructura de árbol – en realidad se implementa como un manifiesto aplanado – pero para nuestros propósitos, es lo suficientemente cerca.

El Directorio de Trabajo

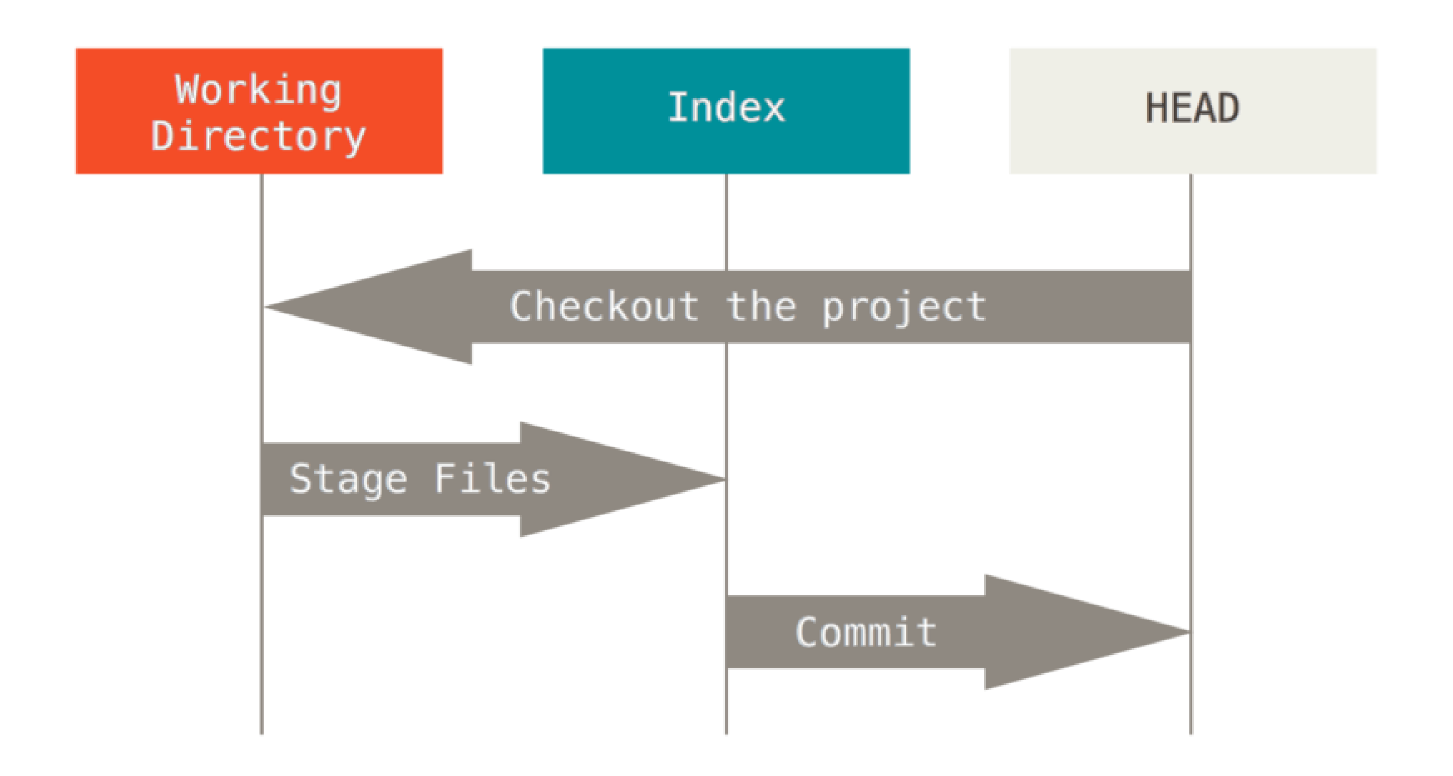
Finalmente, tienes tu directorio de trabajo. Los otros dos árboles almacenan su contenido de manera eficiente pero inconveniente, dentro de la carpeta `.git`. El Directorio de trabajo los descomprime en archivos reales, lo que hace que sea mucho más fácil para usted editarlos. Piense en el Directorio de Trabajo como una **caja de arena**, donde puede probar los cambios antes de enviarlos a su área de ensayo (índice) y luego al historial.

```
$ tree
.
├──── README
├──── Rakefile
└──── lib
    └──── simplegit.rb

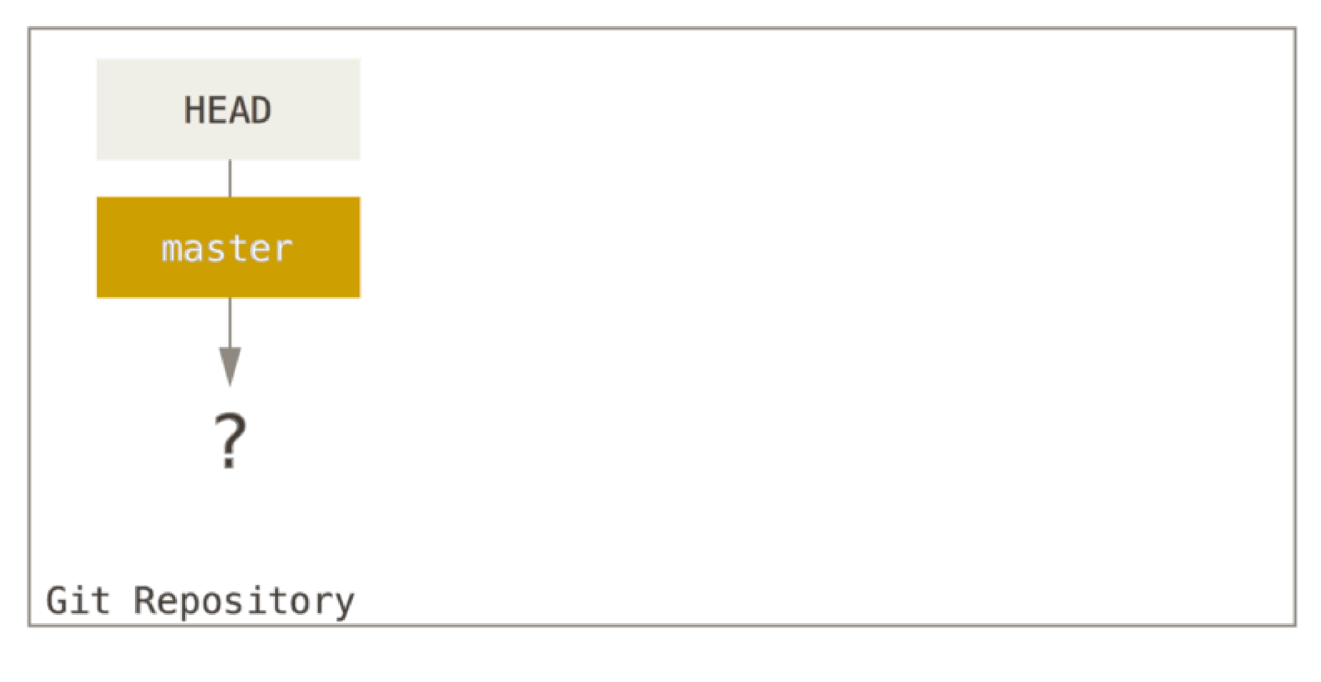
1 directory, 3 files
```

El Flujo de Trabajo

El objetivo principal de Git es registrar instantáneas de su proyecto en estados sucesivamente mejores, mediante la manipulación de estos tres árboles.



Visualicemos este proceso: digamos que ingresa en un nuevo directorio con un solo archivo. Llamaremos a esto **v1** del archivo, y lo indicaremos en azul. Ahora ejecutamos `git init`, que creará un repositorio Git con una referencia HEAD que apunta a una rama no nacida (`master` aún no existe).





En este punto, solo el árbol del Directorio de Trabajo tiene cualquier contenido.

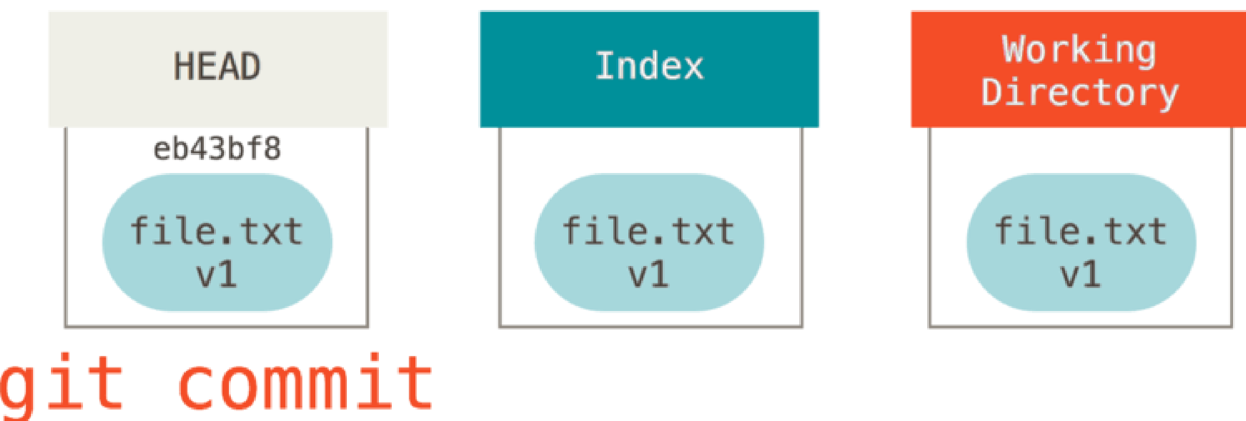
Ahora queremos hacer commit a este archivo, por lo que usamos `git add` para tomar contenido en el directorio de trabajo y copiarlo en el índice.





Luego ejecutamos `git commit`, que toma los contenidos del índice y los guarda como una instantánea permanente, crea un objeto de commit que apunta a esa instantánea y actualiza `master` para apuntar a esa commit.

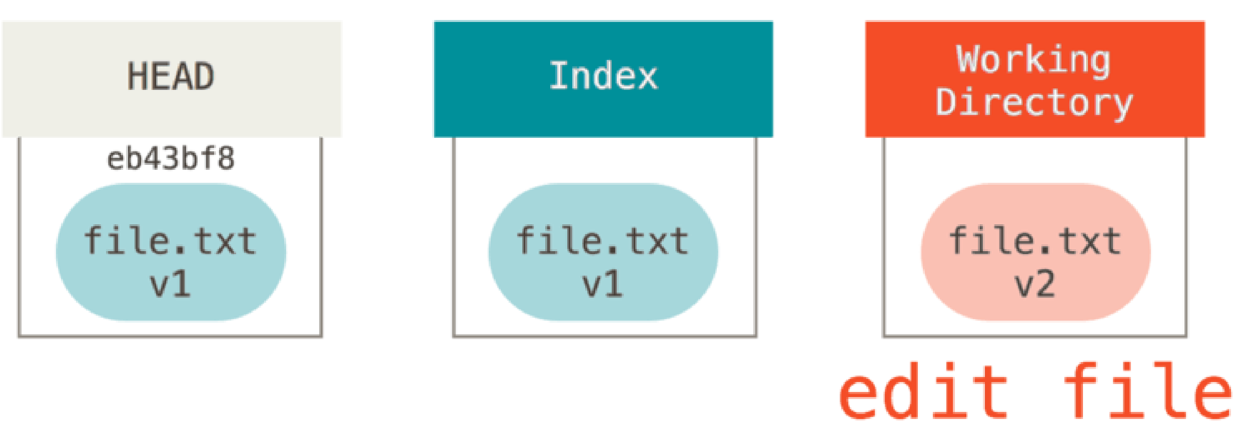




Si ejecutamos `git status`, no veremos ningún cambio, porque los tres árboles son iguales.

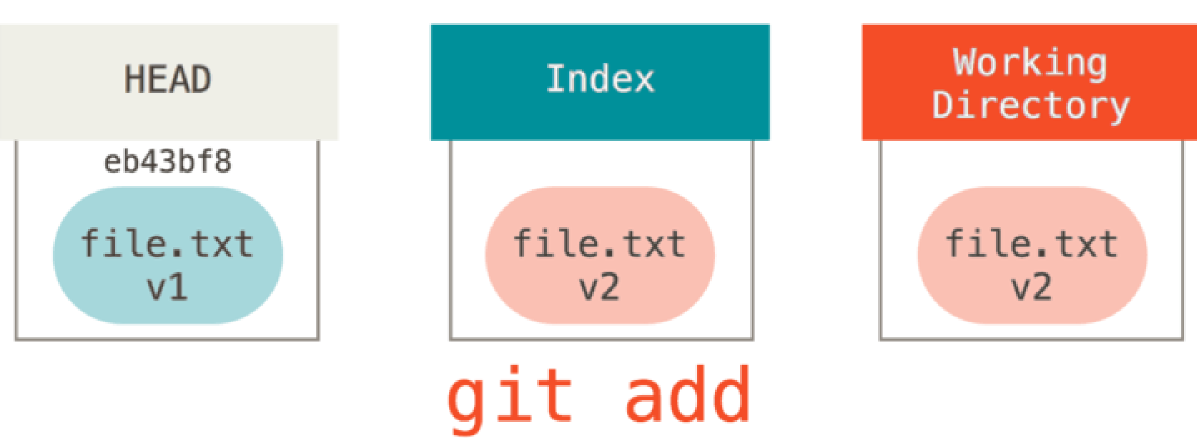
Ahora queremos hacer un cambio en ese archivo y hacerle commit. Pasaremos por el mismo proceso; primero, cambiamos el archivo en nuestro directorio de trabajo. Llamemos a esto **v2** del archivo, y lo indicamos en rojo.



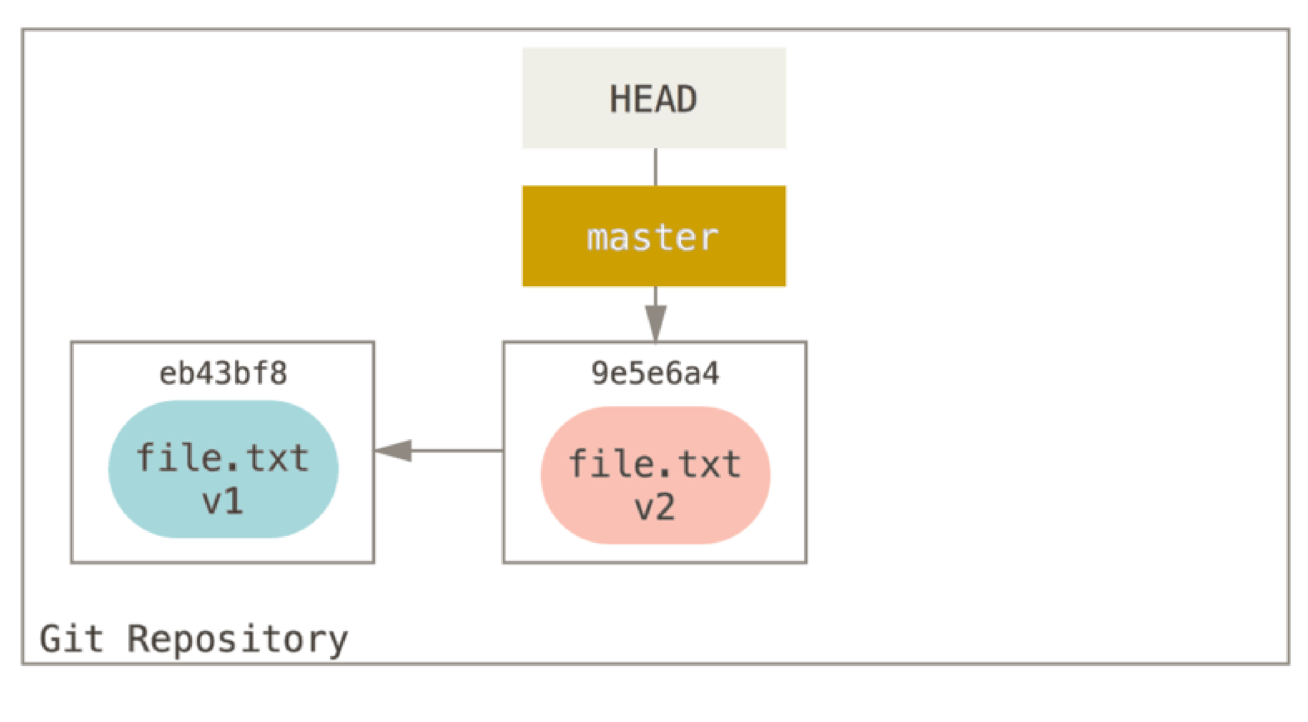


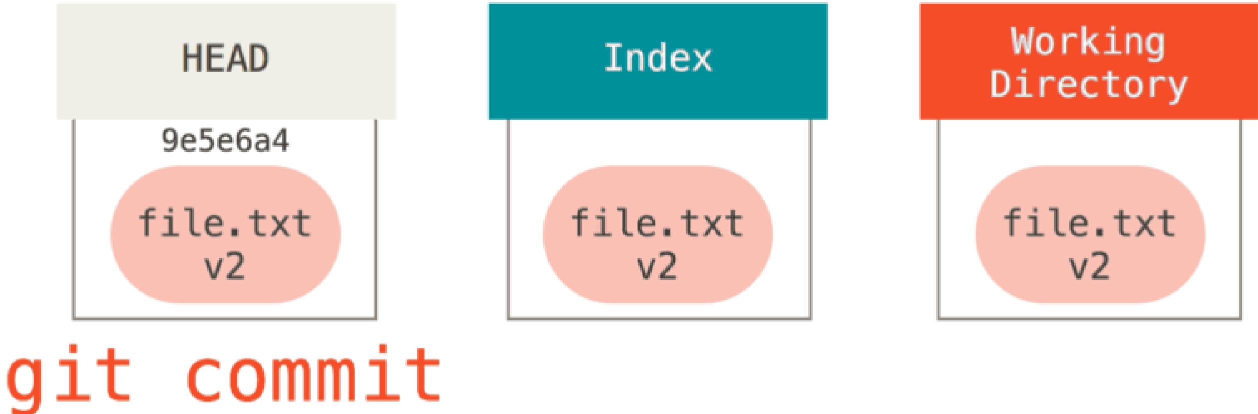
Si ejecutamos `git status` ahora, veremos el archivo en rojo como “Changes not staged for commit” porque esa entrada difiere entre el índice y el directorio de trabajo. A continuación, ejecutamos `git add` para ubicarlo en nuestro índice.





En este punto si ejecutamos `git status` veremos el archivo en verde debajo de “Changes to be committed” porque el Índice y el HEAD difieren – es decir, nuestro siguiente commit propuesta ahora es diferente de nuestra última commit. Finalmente, ejecutamos `git commit` para finalizar el commit.





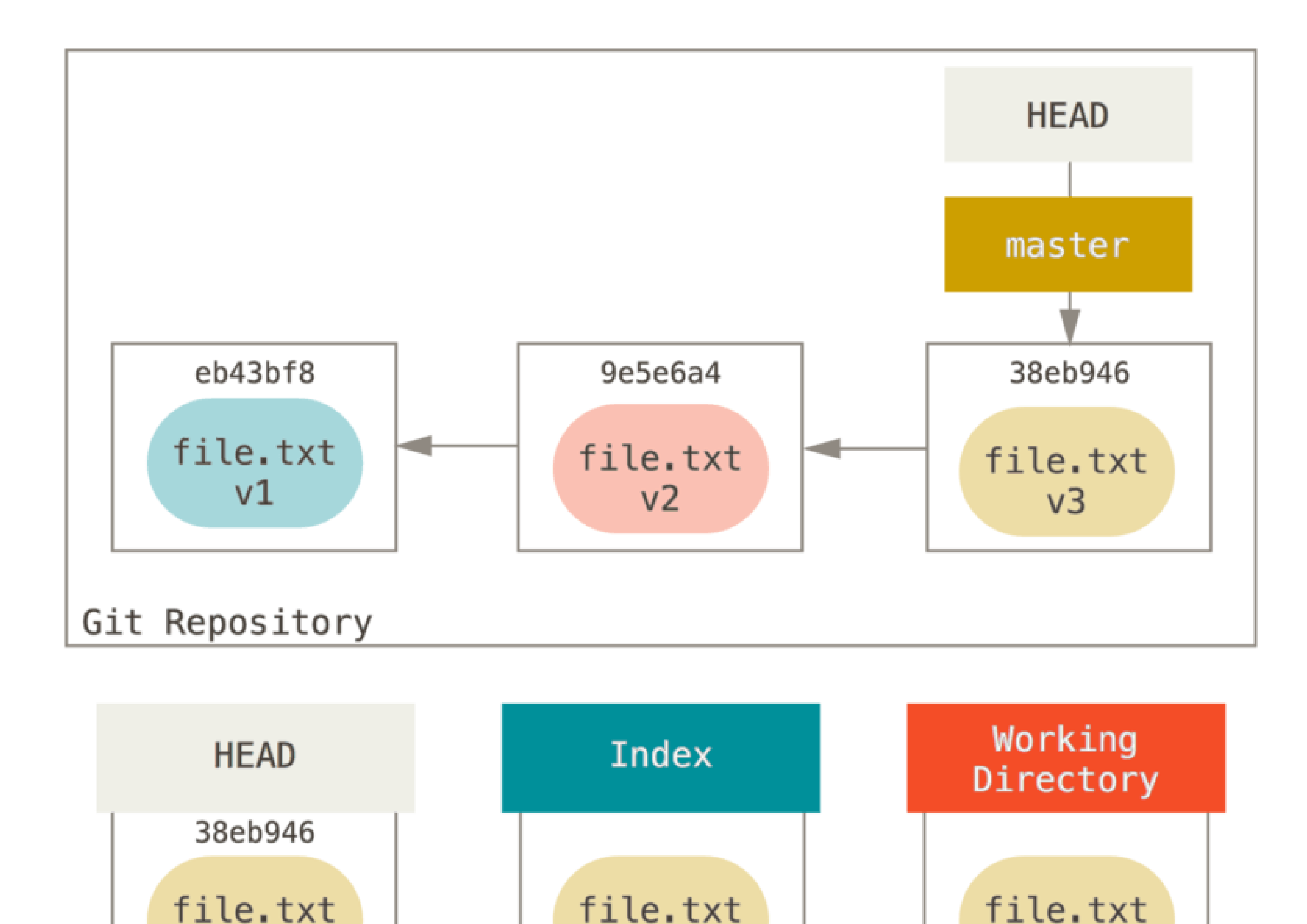
Ahora `git status` no nos dará salida, porque los tres árboles son iguales nuevamente.

El cambio de ramas o la clonación pasa por un proceso similar. Cuando verifica una rama, eso cambia **HEAD** para que apunte a la nueva ref de la rama, rellena su **Índice** con la instantánea de esa confirmación, luego copia los contenidos del **Índice** en su **Directorio de Trabajo**.

El Papel del Reinicio

El comando `reset` tiene más sentido cuando se ve en este contexto.

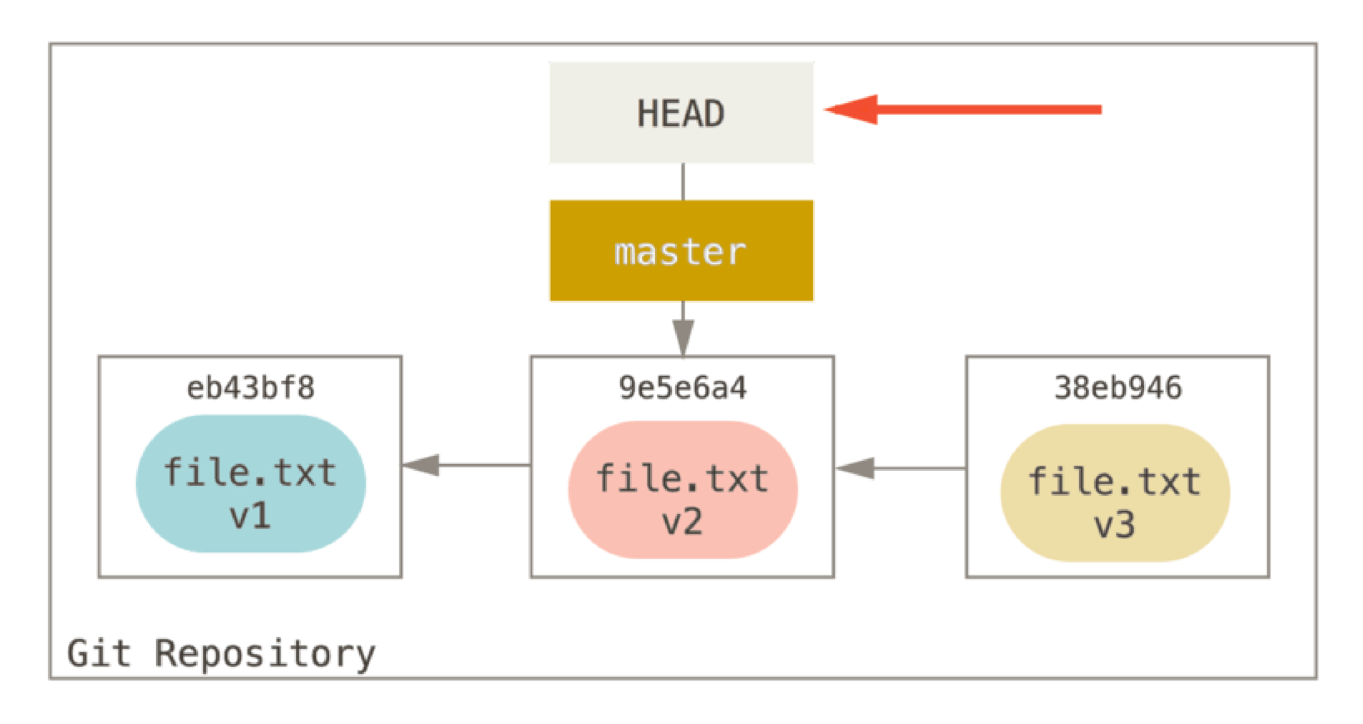
A los fines de estos ejemplos, digamos que hemos modificado `file.txt` de nuevo y lo hemos hecho commit por tercera vez. Entonces ahora nuestra historia se ve así:

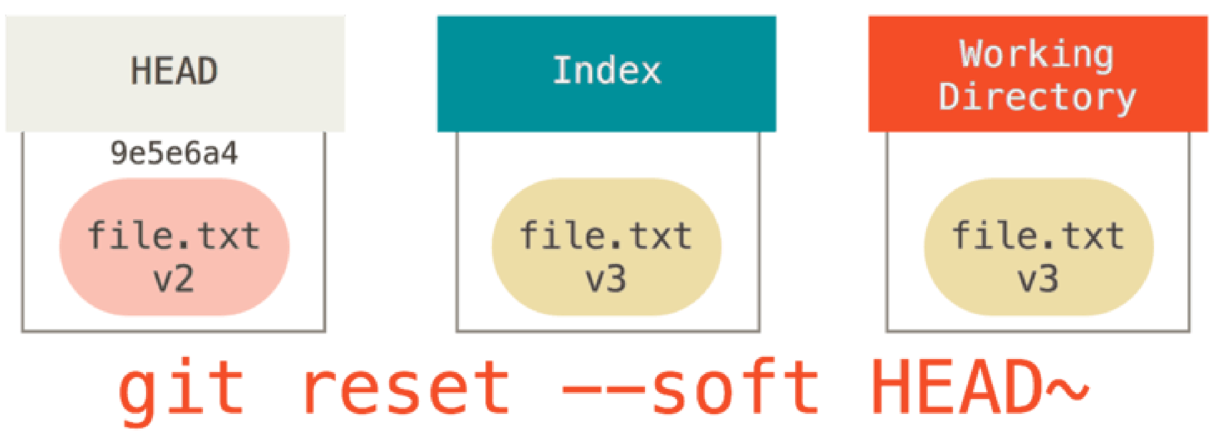


Caminemos ahora a través de exactamente lo que `reset` hace cuando lo llama. Manipula directamente estos tres árboles de una manera simple y predecible. Hace hasta tres operaciones básicas.

Paso 1: mover HEAD

Lo primero que `reset` hará es mover a lo que HEAD apunta. Esto no es lo mismo que cambiar HEAD en sí mismo (que es lo que hace `checkout`); `reset` mueve la rama a la que HEAD apunta. Esto significa que si HEAD está configurado en la rama `master` (es decir, usted está actualmente en la rama `master`), ejecutar `git reset 9e5e64a` comenzará haciendo que `master` apunte a `9e5e64a`.





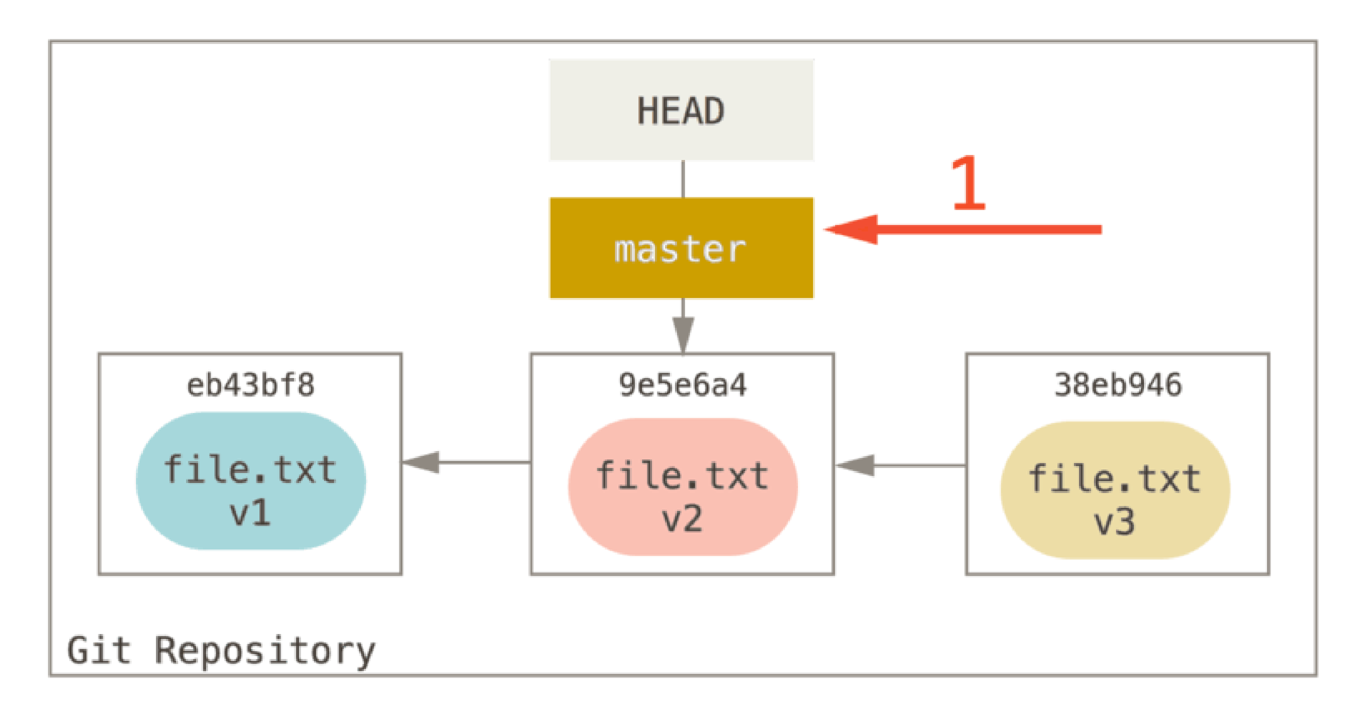
No importa qué forma de `reset` con un commit invoque ustéd, esto es lo primero que siempre intentará hacer. Con `reset --soft`, simplemente se detendrá allí.

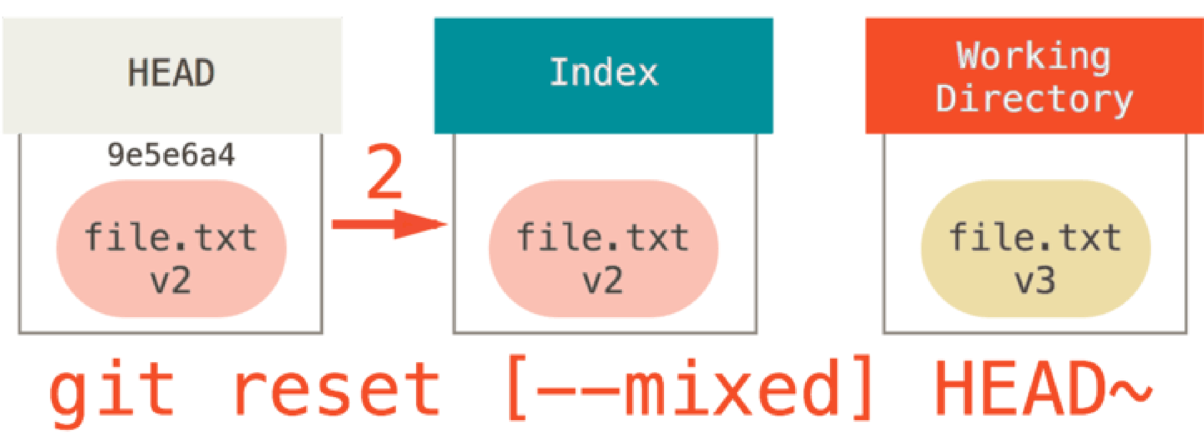
Ahora tómese un segundo para mirar ese diagrama y darse cuenta de lo que sucedió: esencialmente deshizo el último comando `git commit`. Cuando ejecuta `git commit`, Git crea una nueva confirmación y mueve la rama a la que apunta HEAD. Cuando haces `reset` de vuelta a `HEAD~` (el padre de HEAD), está volviendo a colocar la rama donde estaba, sin cambiar el índice o el Directorio de Trabajo. Ahora puedes actualizar el índice y ejecutar `git commit` nuevamente para lograr lo que `git commit --amend` hubiera hecho (ver Cambiando la última confirmación).

Paso 2: Actualizando el índice (--mixed)

Tenga en cuenta que si ejecuta `git status` ahora verá en verde la diferencia entre el Índice y lo que el nuevo HEAD es.

Lo siguiente que `reset` hará es actualizar el Índice con los contenidos de cualquier instantánea que HEAD señale ahora.



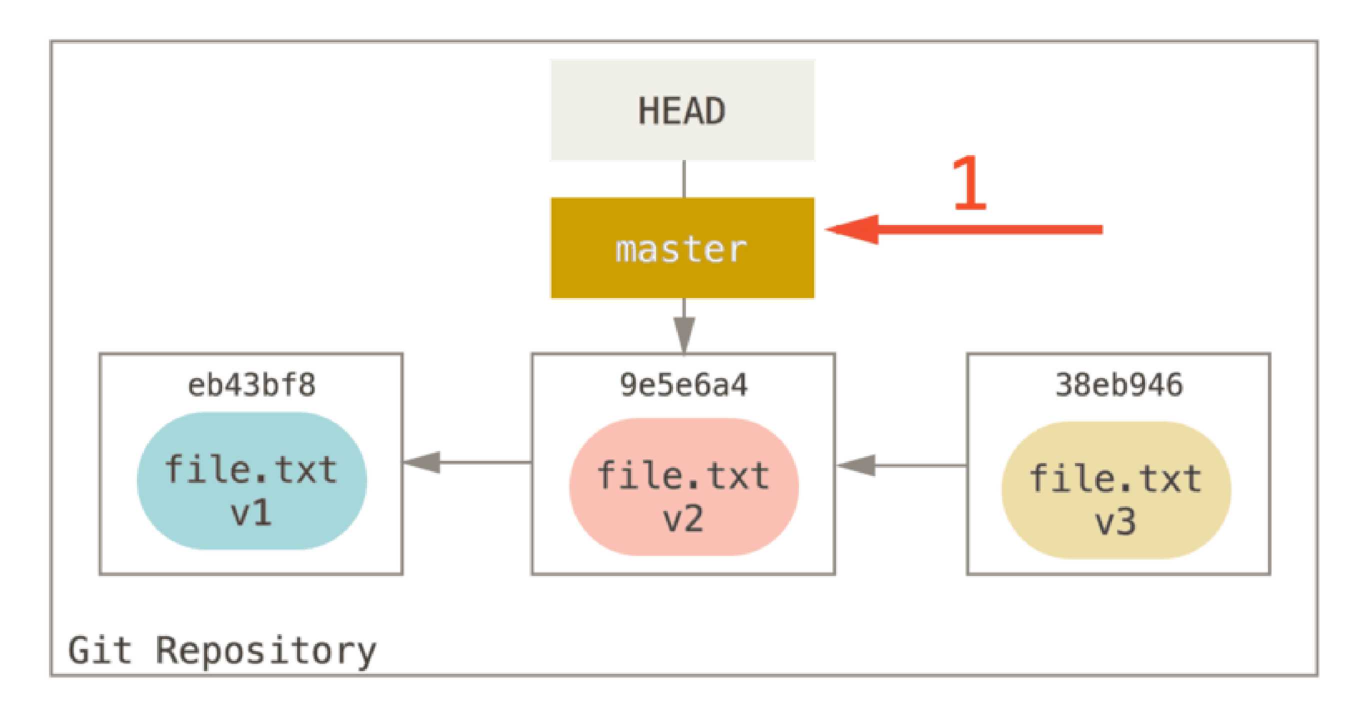


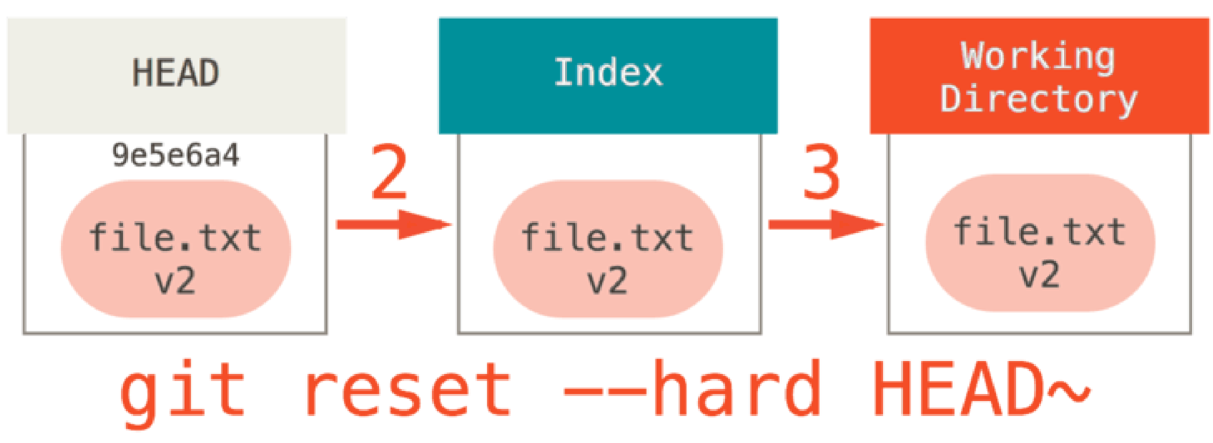
Si especifica la opción `--mixed`, `reset` se detendrá en este punto. Este también es el por defecto, por lo que si no especifica ninguna opción (solo `git reset HEAD~` en este caso), aquí es donde el comando se detendrá.

Ahora tómese otro segundo para mirar ese diagrama y darse cuenta de lo que sucedió: todavía deshizo su último `commit`, pero también hizo *unstaged* de todo. Retrocedió a antes de ejecutar todos sus comandos `git add` y `git commit`.

Paso 3: Actualizar el Directorio de Trabajo (--hard)

Lo tercero que `reset` hará es hacer que el Directorio de Trabajo se parezca al Índice. Si usa la opción `--hard`, continuará en esta etapa.





Entonces, pensemos en lo que acaba de pasar. Deshizo su último commit, los comandos `git add` y `git commit`, **y** todo el trabajo que hizo en su directorio de trabajo.

Es importante tener en cuenta que este indicador (`--hard`) es la única manera de hacer que el comando `reset` sea peligroso, y uno de los pocos casos en que Git realmente destruirá los datos. Cualquier otra invocación de `reset` puede deshacerse fácilmente, pero la opción `--hard` no puede, ya que sobrescribe forzosamente los archivos en el Directorio de Trabajo. En este caso particular, todavía tenemos la versión **v3** de nuestro archivo en una commit en nuestro DB de Git, y podríamos recuperarla mirando nuestro `reflog`, pero si no le hubiéramos hecho commit, Git hubiese sobrescrito el archivo y sería irrecuperable.

Resumen

El comando `reset` sobrescribe estos tres árboles en un orden específico, deteniéndose cuando ustéd le dice:

1. Mueva los puntos HEAD de la rama a *(deténgase aquí si `--soft`)*
2. Haga que el Índice se vea como HEAD *(deténgase aquí a menos que `--hard`)*
3. Haga que el Directorio de Trabajo se vea como el Índice

Reiniciar Con una Ruta

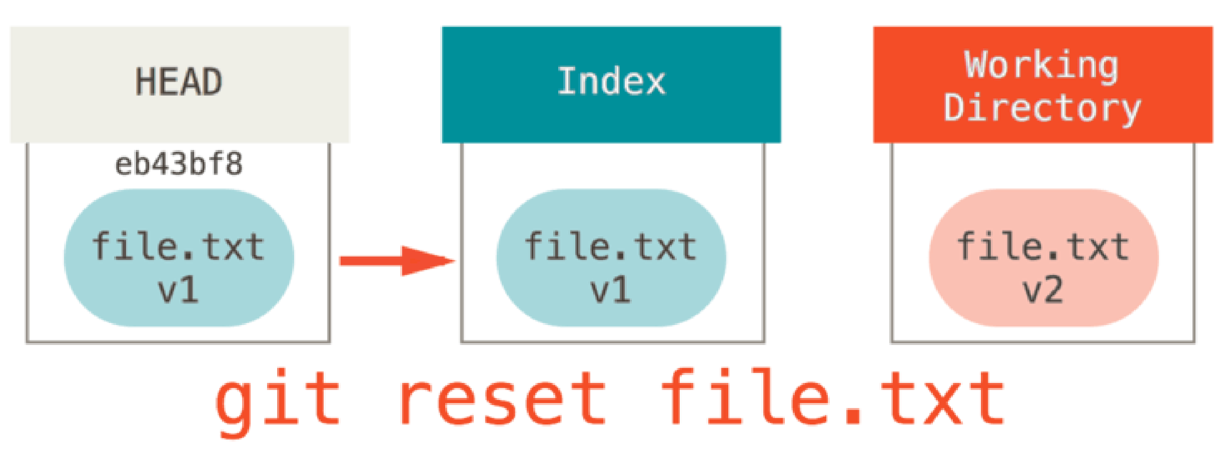
Eso cubre el comportamiento de `reset` en su forma básica, pero también puede proporcionarle una ruta para actuar. Si especifica una ruta, `reset` omitirá el paso 1 y limitará el resto de sus acciones a un archivo o conjunto específico de archivos. Esto realmente tiene sentido – HEAD es solo un puntero, y no se puede apuntar a una parte de una commit y parte de otra. Pero el índice y el Directorio de Trabajo *pueden* actualizarse parcialmente, por lo que el reinicio continúa con los pasos 2 y 3.

Entonces, supongamos que ejecutamos `git reset file.txt`. Este formulario (ya que no especificó un commit SHA-1 o una rama, y no especificó `--soft` o `--hard`) es una abreviatura de `git reset --mixed HEAD file.txt`, la cual hará:

1. Mueva los puntos HEAD de la rama a *(omitido)*
2. Haga que el Índice se vea como HEAD *(deténgase aquí)*

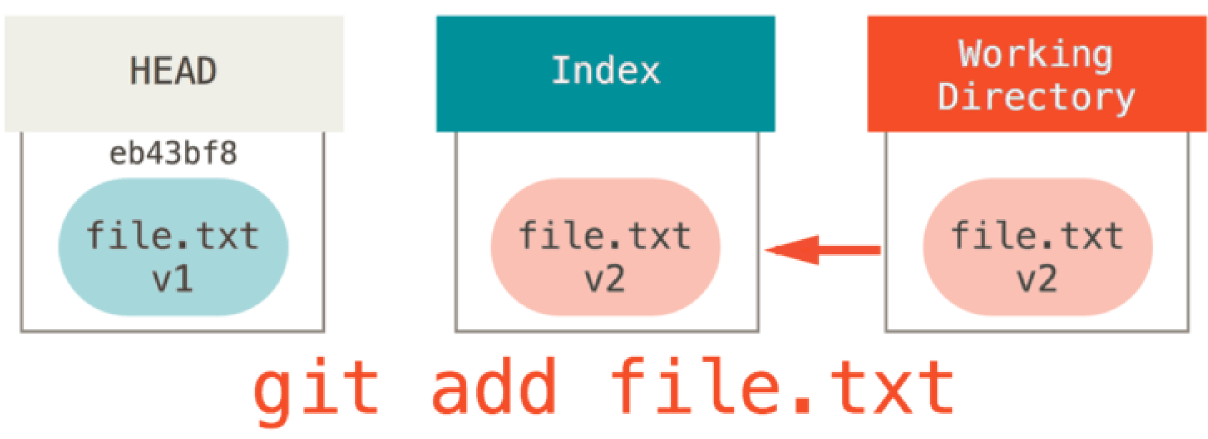
Por lo tanto, básicamente solo copia `archivo.txt` de HEAD al Índice.





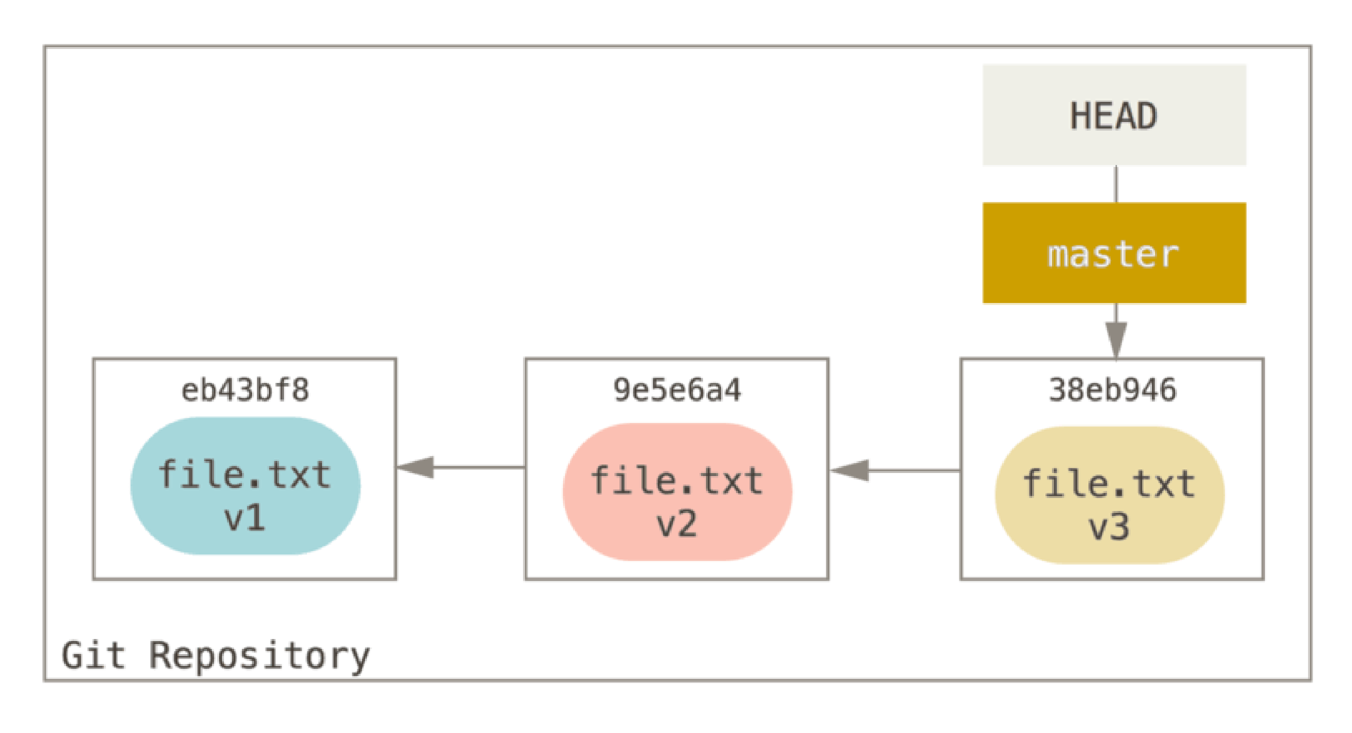
Esto tiene el efecto práctico de hacer *unstaging* al archivo. Si miramos el diagrama para ese comando y pensamos en lo que hace `git add`, son exactamente opuestos.





Esta es la razón por la cual el resultado del comando `git status` sugiere que ejecute esto para descentralizar un archivo. (Consulte Deshacer un Archivo Preparado para más sobre esto).

Igualmente podríamos no permitir que Git suponga que queríamos "extraer los datos de HEAD" especificando una commit específica para extraer esa versión del archivo. Simplemente ejecutaríamos algo como `git reset eb43bf file.txt`.





Esto efectivamente hace lo mismo que si hubiéramos revertido el contenido del archivo a **v1** en el Directorio de Trabajo, ejecutáramos `git add` en él, y luego lo revertimos a **v3** nuevamente (sin tener que ir a través de todos esos pasos) Si ejecutamos `git commit` ahora, registrará un cambio que revierte ese archivo de vuelta a **v1**, aunque nunca más lo volvimos a tener en nuestro Directorio de Trabajo.

También es interesante observar que, como `git add`, el comando `reset` aceptará una opción `--patch` para hacer unstage del contenido en una base hunk-by-hunk. Por lo tanto, puede hacer unstage o revertir el contenido de forma selectiva.

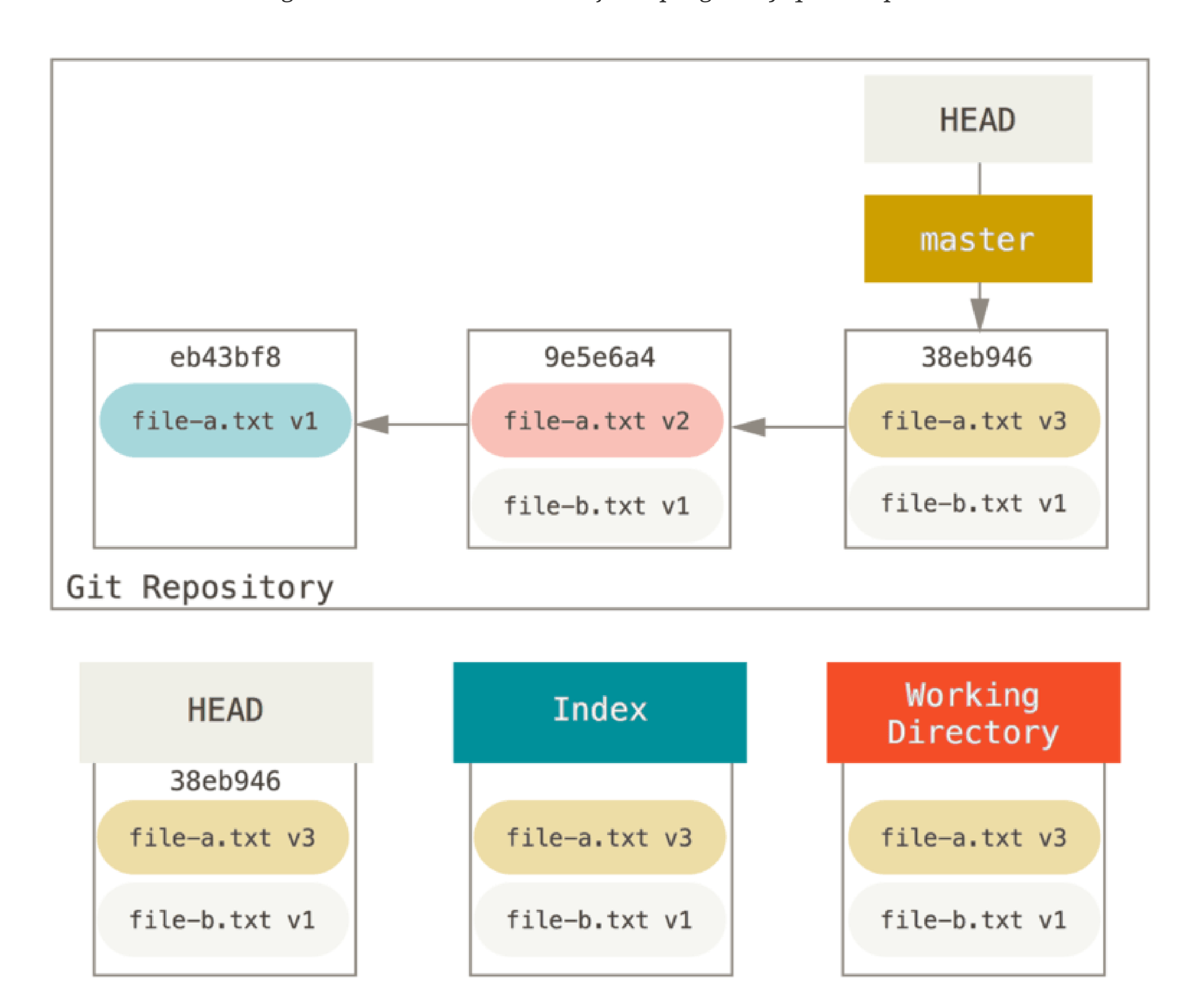
Aplastando

Veamos cómo hacer algo interesante con este poder recién descubierto – aplastando commits.

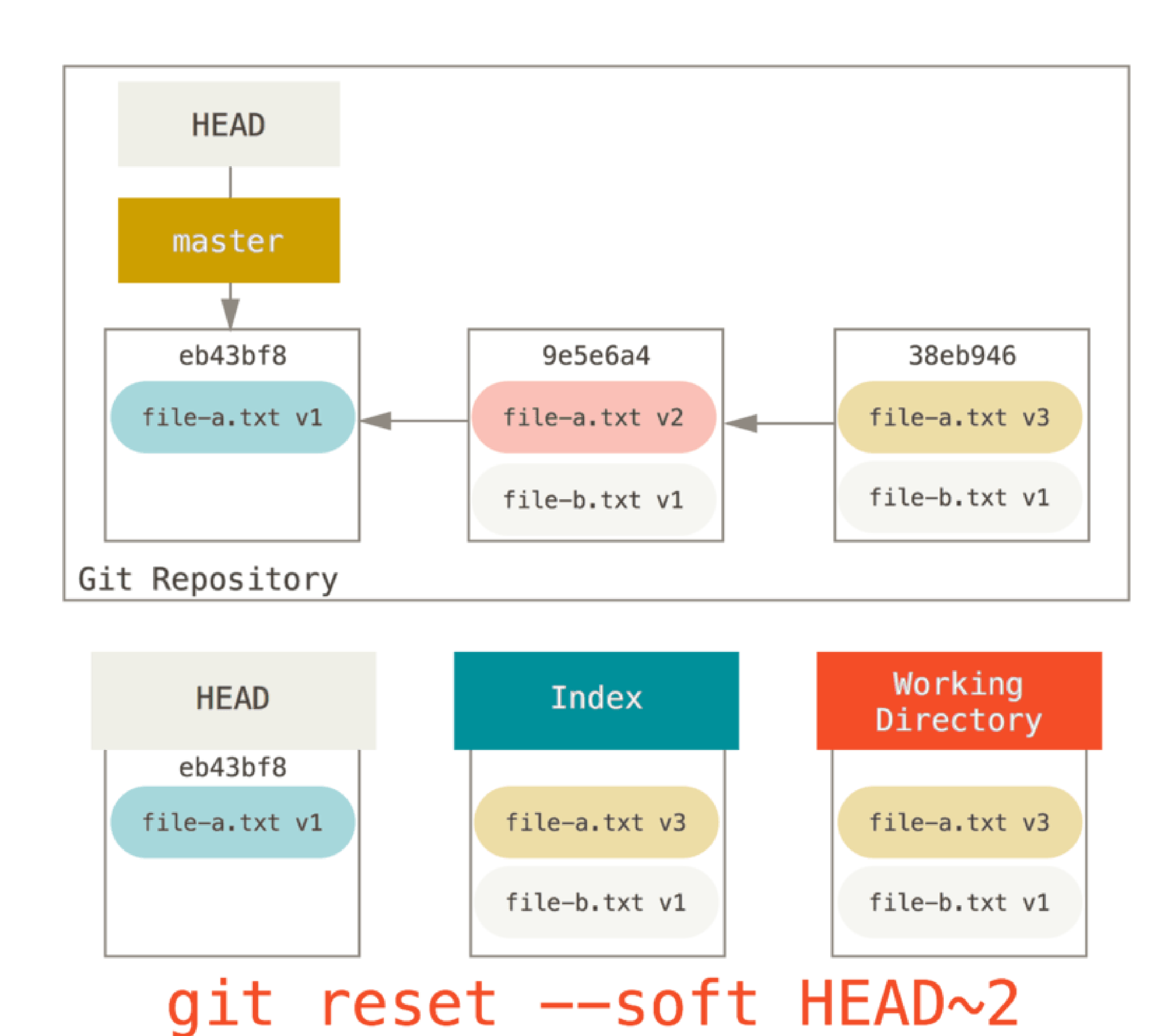
Supongamos que tiene una serie de confirmaciones con mensajes como "oops.", "WIP" y "se olvidó de este archivo". Puedes usar `reset` para aplastarlos rápida y fácilmente en una sola confirmación que lo hace ver realmente inteligente. (Aplastando muestra otra forma de hacerlo, pero en este ejemplo es más simple usar `reset`.)

Supongamos que tiene un proyecto en el que la primera commit tiene un archivo, la segunda

commit agregó un nuevo archivo y cambió la primera, y la tercera commit cambió el primer archivo otra vez. La segunda commit fué un trabajo en progreso y quiere aplastarlo.



Puede ejecutar `git reset --soft HEAD~2` para mover la rama HEAD a una commit anterior (la primera commit que desea mantener):



Y luego simplemente ejecute `git commit` nuevamente:



Ahora puede ver que su historial alcanzable, la historia que empujaría, ahora parece que tuvo una commit con `archivo-a.txt` v1, luego un segundo que ambos modificaron `archivo-a.txt` a v3 y agregaron `archivo-b.txt`. La commit con la versión v2 del archivo ya no está en el historial.

Echale Un vistazo

Finalmente, puede preguntarse cuál es la diferencia entre `checkout` y `reset`. Al igual que `reset`, `checkout` manipula los tres árboles, y es un poco diferente dependiendo de si le da al comando una ruta de archivo o no.

Sin Rutas

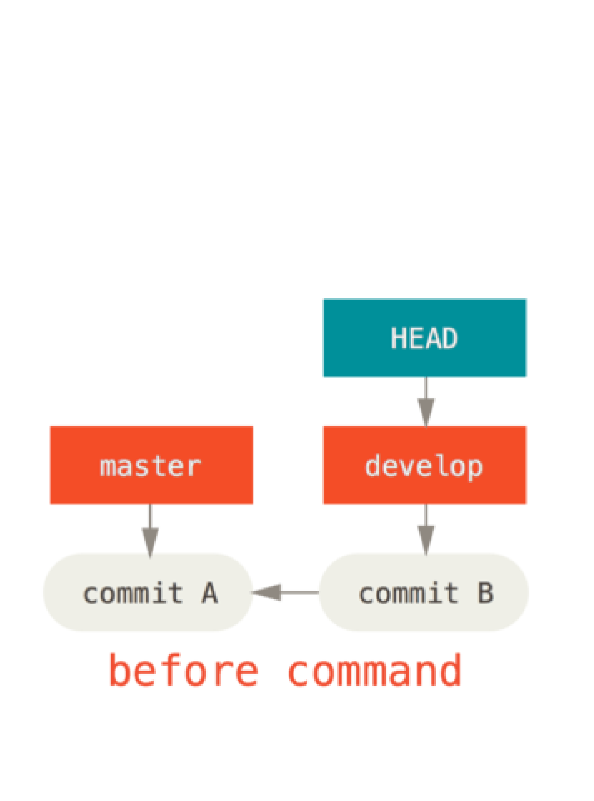
Ejecutar `git checkout [branch]` es bastante similar a ejecutar `git reset --hard [branch]` porque actualiza los tres árboles para que se vea como `[branch]`, pero hay dos diferencias importantes.

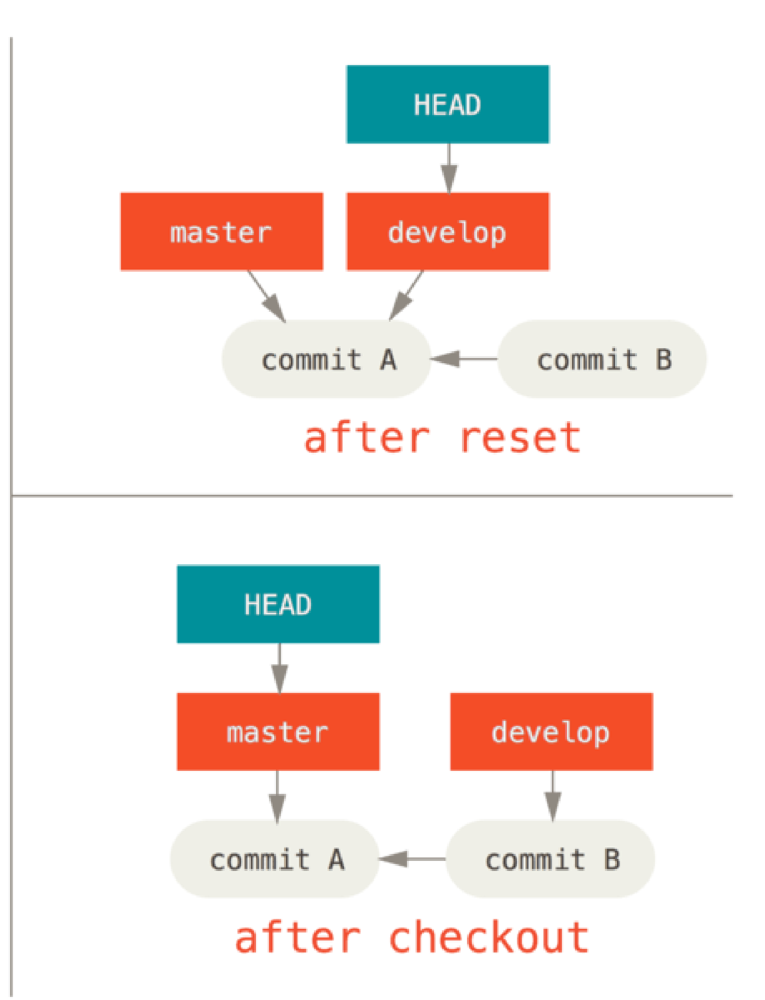
Primero, a diferencia de `reset --hard`, `checkout` está directorio-de-trabajo seguro; Verificará para asegurarse de que no está volando los archivos que tienen cambios en ellos. En realidad, es un poco más inteligente que eso – intenta hacer una fusión trivial en el Directorio de Trabajo, por lo que todos los archivos que *no haya* cambiado serán actualizados. `reset --hard`, por otro lado, simplemente reemplazará todo en general sin verificar.

La segunda diferencia importante es cómo actualiza HEAD. Donde `reset` moverá la rama a la que HEAD apunta, `checkout` moverá HEAD para señalar otra rama.

Por ejemplo, digamos que tenemos las ramas `master` y `develop` que apuntan a diferentes commits, y actualmente estamos en `develop` (así que HEAD lo señala). Si ejecutamos `git reset master`, `develop` ahora apuntará a la misma commit que `master`. Si en cambio ejecutamos `git checkout master`, `develop` no se mueve, HEAD sí lo hace. HEAD ahora apuntará a `master`.

Entonces, en ambos casos estamos moviendo HEAD para apuntar a la commit A, pero *cómo* lo hacemos es muy diferente. `reset` moverá los puntos HEAD de la rama a, `checkout` mueve el mismo HEAD.





Con Rutas

La otra forma de ejecutar `checkout` es con una ruta de archivo, que, como `reset`, no mueva HEAD. Es como `git reset [branch] file` en que actualiza el índice con ese archivo en esa commit, pero también sobrescribe el archivo en el directorio de trabajo. Sería exactamente como `git reset --hard [branch] file` (si `reset` le permitiera ejecutar eso) - no está directorio-de-trabajo seguro, y no mueve a HEAD.

Además, al igual que `git reset` y `git add`, `checkout` aceptará una opción `--patch` para permitirle revertir selectivamente el contenido del archivo sobre una base hunk-by-hunk.

Resumen

Esperemos que ahora entienda y se sienta más cómodo con el comando `reset`, pero probablemente todavía esté un poco confundido acerca de cómo exactamente difiere de `checkout` y posiblemente no pueda recordar todas las reglas de las diferentes invocaciones.

Aquí hay una hoja de trucos para cuales comandos afectan a cuales árboles. La columna “HEAD” lee “REF” si ese comando mueve la referencia (rama) a la que HEAD apunta, y “HEAD” si mueve al mismo HEAD. Esta función calculará la ciudad y el país del visitante desde la dirección IP. Para activar esta función, debes descargar la base de datos GeoIP y configurar la clave de la API de Google. Por favor, vea "Más información"

	HEAD	Índice	Dirtrabajo	DT Seguro?
Nivl de la Commit				
`reset --soft [commit]`	REF	NO	NO	SI
`reset [commit]`	REF	SI	NO	SI
`reset --hard [commit]`	REF	SI	SI	**NO**
`checkout [commit]`	HEAD	SI	SI	SI
Nivel de Archivo				
`reset (commit) [file]`	NO	SI	NO	SI
`checkout (commit) [file]`	NO	SI	SI	**NO**

Fusión Avanzada

La fusión en Git suele ser bastante fácil. Dado que Git facilita la fusión de otra rama varias veces, significa que puede tener una rama de larga duración, pero puede mantenerla actualizada sobre la marcha, resolviendo pequeños conflictos a menudo, en lugar de sorprenderse por un conflicto enorme en el final de la serie.

Sin embargo, a veces ocurren conflictos engañosos. A diferencia de otros sistemas de control de versiones, Git no intenta ser demasiado listo para fusionar la resolución de conflictos. La filosofía de Git es ser inteligente para determinar cuándo una resolución de fusión no es ambigua, pero si hay un conflicto, no intenta ser inteligente para resolverlo automáticamente. Por lo tanto, si espera demasiado para fusionar dos ramas que divergen rápidamente, puede encontrarse algunos problemas.

En esta sección, veremos cuáles podrían ser algunos de esos problemas y qué herramientas le dará Git para ayudarlo a manejar estas situaciones más engañosas. También cubriremos algunos de los diferentes tipos de fusión no estándar que puede hacer, y también veremos cómo deshacerse de las fusiones que ha realizado.

Conflictos de Fusión

Si bien cubrimos algunos conceptos básicos para resolver conflictos de fusión en Principales Conflictos que Pueden Surgir en las Fusiones, para conflictos más complejos, Git proporciona algunas herramientas para ayudarlo a descubrir qué está sucediendo y cómo lidiar mejor con el conflicto.

En primer lugar, si es posible, intente asegurarse de que su directorio de trabajo esté limpio antes de realizar una fusión que pueda tener conflictos. Si tiene un trabajo en progreso, hágale commit a una rama temporal o stash. Esto hace que puedas deshacer **cualquier cosa** que intentes aquí. Si tiene cambios no guardados en su directorio de trabajo cuando intenta fusionarlos, algunos de estos consejos pueden ayudarlo a perder ese trabajo.

Veamos un ejemplo muy simple. Tenemos un archivo Ruby super simple que imprime *hello world*.

```
#! /usr/bin/env ruby

def hello
  puts 'hello world'
end

hello()
```

En nuestro repositorio, creamos una nueva rama llamada `whitespace` y procedemos a cambiar todas las terminaciones de línea de Unix a terminaciones de línea de DOS, esencialmente cambiando cada línea del archivo, pero solo con espacios en blanco. Luego cambiamos la línea "hello world" a "hello mundo".

```
$ git checkout -b whitespace
Switched to a new branch 'whitespace'

$ unix2dos hello.rb
unix2dos: converting file hello.rb to DOS format ...
$ git commit -am 'converted hello.rb to DOS'
[whitespace 3270f76] converted hello.rb to DOS
 1 file changed, 7 insertions(+), 7 deletions(-)

$ vim hello.rb
$ git diff -w
diff --git a/hello.rb b/hello.rb
index ac51efd..e85207e 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,7 @@
 #! /usr/bin/env ruby

 def hello
-  puts 'hello world'
+  puts 'hello mundo'^M
 end

 hello()

$ git commit -am 'hello mundo change'
[whitespace 6d338d2] hello mundo change
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Ahora volvemos a nuestra rama `master` y agregamos cierta documentación para la función.

```
$ git checkout master
Switched to branch 'master'

$ vim hello.rb
$ git diff
diff --git a/hello.rb b/hello.rb
index ac51efd..36c06c8 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
 #! /usr/bin/env ruby

+# prints out a greeting
 def hello
   puts 'hello world'
 end

$ git commit -am 'document the function'
[master bec6336] document the function
 1 file changed, 1 insertion(+)
```

Ahora tratamos de fusionarnos en nuestra rama `whitespace` y tendremos conflictos debido a los cambios en el espacio en blanco.

```
$ git merge whitespace
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Automatic merge failed; fix conflicts and then commit the result.
```

Abortar una Fusión

Ahora tenemos algunas opciones. Primero, cubramos cómo salir de esta situación. Si tal vez no esperabas conflictos y aún no quieres lidiar con la situación, simplemente puedes salir de la fusión con `git merge --abort`.

```
$ git status -sb
## master
UU hello.rb

$ git merge --abort

$ git status -sb
## master
```

La opción `git merge --abort` intenta volver a su estado antes de ejecutar la fusión. Los únicos casos en los que podría no ser capaz de hacer esto a la perfección serían si hubiera realizado cambios sin stash, no confirmados en su directorio de trabajo cuando lo ejecutó, de lo contrario, debería

funcionar bien.

Si por alguna razón te encuentras en un estado horrible y solo quieres comenzar de nuevo, también puedes ejecutar `git reset --hard HEAD` o donde quieras volver. Recuerda una vez más que esto hará volar tu directorio de trabajo, así que asegúrate de no querer ningún cambio allí.

Ignorando el Espacio en Blanco

En este caso específico, los conflictos están relacionados con el espacio en blanco. Sabemos esto porque el caso es simple, pero también es muy fácil decirlo en casos reales al analizar el conflicto porque cada línea se elimina por un lado y se agrega nuevamente por el otro. De manera predeterminada, Git ve que todas estas líneas están siendo modificadas, por lo que no puede fusionar los archivos.

Sin embargo, la estrategia de combinación predeterminada puede tomar argumentos, y algunos de ellos son acerca de ignorar adecuadamente los cambios del espacio en blanco. Si ve que tiene muchos problemas con espacios en blanco en una combinación, simplemente puede cancelarla y volverla a hacer, esta vez con `-Xignore-all-space` o` -Xignore-space-change`. La primera opción ignora los cambios en cualquier **cantidad** de espacios en blanco existentes, el segundo ignora por completo todos los cambios de espacios en blanco.

```
$ git merge -Xignore-all-space whitespace
Auto-merging hello.rb
Merge made by the 'recursive' strategy.
 hello.rb | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Dado que en este caso, los cambios reales del archivo no eran conflictivos, una vez que ignoramos los cambios en los espacios en blanco, todo se fusiona perfectamente.

Esto es un salvavidas si tienes a alguien en tu equipo a quien le gusta ocasionalmente reformatear todo, desde espacios hasta pestañas o viceversa.

Re-fusión Manual de Archivos

Aunque Git maneja muy bien el preprocesamiento de espacios en blanco, hay otros tipos de cambios que quizás Git no pueda manejar de manera automática, pero que son correcciones de secuencias de comandos. Como ejemplo, imaginemos que Git no pudo manejar el cambio en el espacio en blanco y que teníamos que hacerlo a mano.

Lo que realmente tenemos que hacer es ejecutar el archivo que intentamos fusionar a través de un programa `dos2unix` antes de intentar fusionar el archivo. Entonces, ¿cómo haríamos eso?

Primero, entramos en el estado de conflicto de la fusión. Luego queremos obtener copias de mi versión del archivo, su versión (de la rama en la que nos estamos fusionando) y la versión común (desde donde ambos lados se bifurcaron). Entonces, queremos arreglar su lado o nuestro lado y volver a intentar la fusión nuevamente solo para este único archivo.

Obtener las tres versiones del archivo es bastante fácil. Git almacena todas estas versiones en el

índice bajo “etapas”, cada una de las cuales tiene números asociados. La etapa 1 es el ancestro común, la etapa 2 es su versión y la etapa 3 es de la `MERGE_HEAD`, la versión en la que se está fusionando (“suya”).

Puede extraer una copia de cada una de estas versiones del archivo en conflicto con el comando `git show` y una sintaxis especial.

```
$ git show :1:hello.rb > hello.common.rb
$ git show :2:hello.rb > hello.ours.rb
$ git show :3:hello.rb > hello.theirs.rb
```

Si quieres ponerte un poco más intenso, también puedes usar el comando de plomería `ls-files -u` para obtener el verdadero SHA-1s de las manchas de Git para cada uno de los archivos.

```
$ git ls-files -u
100755 ac51efdc3df4f4fd328d1a02ad05331d8e2c9111 1	hello.rb
100755 36c06c8752c78d2aff89571132f3bf7841a7b5c3 2	hello.rb
100755 e85207e04dfdd5eb0a1e9febbc67fd837c44a1cd 3	hello.rb
```

El `:1:hello.rb` es solo una clave para buscar esa mancha SHA-1.

Ahora que tenemos el contexto de estas tres etapas en nuestro directorio de trabajo, manualmente podemos arreglar los de ellos para solucionar los problemas de espacios en blanco y volver a fusionar el archivo con el poco conocido comando `git merge-file` que hace exactamente eso.

```
$ dos2unix hello.theirs.rb
dos2unix: converting file hello.theirs.rb to Unix format ...

$ git merge-file -p \
    hello.ours.rb hello.common.rb hello.theirs.rb > hello.rb

$ git diff -w
diff --cc hello.rb
index 36c06c8,e85207e..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,8 -1,7 +1,8 @@@
  #! /usr/bin/env ruby

 +# prints out a greeting
  def hello
-   puts 'hello world'
+   puts 'hello mundo'
  end

  hello()
```

En este punto, hemos agradablemente fusionado el archivo. De hecho, esto en realidad funciona mejor que la opción de `ignore-all-space` porque esto realmente soluciona los cambios de los espacios en blanco antes de la fusión, en lugar de simplemente ignorarlo. En la fusión `ignore-all-space, realmente, terminamos con unas pocas líneas con finales de línea DOS, haciendo que las cosas se mezclen.

Si quieres tener una idea antes de finalizar este compromiso sobre qué había cambiado en realidad entre un lado y el otro, puedes pedirle a `git diff` que compare qué hay en tu directorio de trabajo que estas a punto de comprometer como resultado de la fusión a cualquiera de estas etapas. Vamos a través de todas ellas.

Para comparar el resultado con lo que tenías en tu rama antes de la fusión, en otra palabras, para ver lo que tu fusión insertó, puedes correr `git diff --ours`

```
$ git diff --ours
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index 36c06c8..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -2,7 +2,7 @@

 # prints out a greeting
 def hello
-  puts 'hello world'
+  puts 'hello mundo'
 end

 hello()
```

Así podemos observar fácilmente lo que sucedió en nuestra rama, lo que en realidad estamos insertando a este archivo con esta fusión, está cambiando solamente esa línea.

Si queremos ver cómo el resultado de la fusión difiere de lo que estaba del otro lado, puedes correr `git diff --theirs`. En este y el siguiente ejemplo, tenemos que usar `-w para despojarse de los espacios en blanco porque lo estamos comparando con lo que está en Git, no con nuestro archivo limpio `hello.theirs.rb

```
$ git diff --theirs -w
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index e85207e..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
 #! /usr/bin/env ruby

+# prints out a greeting
 def hello
   puts 'hello mundo'
 end
```

Finalmente, puedes observar cómo el archivo ha cambiado desde ambos lados con `git diff --base`.

```
$ git diff --base -w
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index ac51efd..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,8 @@
 #! /usr/bin/env ruby

+# prints out a greeting
 def hello
-  puts 'hello world'
+  puts 'hello mundo'
 end

 hello()
```

En este punto podemos usar el comando `git clean` para limpiar los archivos sobrantes que creamos para hacer el manual de la fusión, pero que ya no necesitamos.

```
$ git clean -f
Removing hello.common.rb
Removing hello.ours.rb
Removing hello.theirs.rb
```

Revisando Los Conflictos

Tal vez en este punto no estemos felices con la resolución por alguna razón, o quizás manualmente editando uno o ambos lados todavía no funciona como es debido y necesitamos más contexto.

Cambiemos el ejemplo un poco. Para este ejemplo, tenemos dos ramas de larga vida las cuales cada

una tiene unos pocos cometidos en ellas aparte de crear un contenido de conflicto legítimo cuando es fusionado.

```
$ git log --graph --oneline --decorate --all
* f1270f7 (HEAD, master) update README
* 9af9d3b add a README
* 694971d update phrase to hola world
| * e3eb223 (mundo) add more tests
| * 7cff591 add testing script
| * c3ffff1 changed text to hello mundo
|/
* b7dcc89 initial hello world code
```

Ahora tenemos tres cometidos únicos que viven solo en la rama `principal` y tres otras que viven en la rama `mundo`. Si intentamos fusionar la rama mundo, generamos un conflicto.

```
$ git merge mundo
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Automatic merge failed; fix conflicts and then commit the result.
```

Nos gustaría ver cuál es el conflicto de fusión. Si abrimos el archivo, veremos algo así:

```
#! /usr/bin/env ruby

def hello
<<<<<<< HEAD
  puts 'hola world'
=======
  puts 'hello mundo'
>>>>>>> mundo
end

hello()
```

Ambos lados de la fusión han añadido contenido a este archivo, pero algunos de los cometidos han modificado el archivo en el mismo lugar que causó el conflicto.

Exploremos un par de herramientas que ahora tienes a tu disposición para determinar cómo el conflicto resultó ser. Tal vez no es tan obvio qué tan exactamente deberías solucionar este problema. Necesitas más contexto.

Una herramienta útil es `git checkout` con la opción '--conflicto'. Esto revisará el archivo de nuevo y reemplazará los marcadores de conflicto de la fusión. Esto puede ser útil si quieres reiniciar los marcadores y tratar de resolverlos de nuevo.

Puedes pasar `--conflict` en lugar de `diff3` o `merge` (lo que es por defecto). Si pasas `diff3`, Git usará

una versión un poco diferente de marcadores de conflicto, no solo dándote “ours” versión y la versión de “theirs”, sino también la versión “base” en línea para darte más contexto.

```
$ git checkout --conflict=diff3 hello.rb
```

Una vez que corremos eso, en su lugar el archivo se verá así:

```
#! /usr/bin/env ruby

def hello
<<<<<<< ours
  puts 'hola world'
||||||| base
  puts 'hello world'
=======
  puts 'hello mundo'
>>>>>>> theirs
end

hello()
```

Si este formato es de tu agrado, puedes configurarlo como defecto para futuros conflictos de fusión al colocar el `merge.conflictstyle` configurándolo a `diff3`.

```
$ git config --global merge.conflictstyle diff3
```

El comando `git checkout` puede también tomar la opción de `--ellos`o la `--nuestra`, lo cual puede ser una manera mucho más rápida de escoger un lado o el otro sin tener que fusionar las cosas en lo absoluto.

Esto puede ser particularmente útil para conflictos de archivos binarios donde simplemente puedes escoger un lado, o donde solo quieres fusionar ciertos archivos desde otra rama – puedes hacer la fusión y luego revisar ciertos archivos de un lado o del otro antes de comprometerlos

Registro de Fusión

Otra herramienta útil al resolver conflictos de fusión es `git log`. Esto puede ayudarte a tener contexto a lo que pudo haber contribuido a los conflictos. Revisando un poco de historial para recordar por qué dos líneas de desarrollo estaban tocando el mismo código de área puede ser muy útil algunas veces.

Para obtener una lista completa de cometidos únicos que fueron incluidos en cualquiera de las ramas involucradas en esta fusión, podemos usar la sintaxis “triple dot” (triple punto) que aprendimos en Tres puntos.

```
$ git log --oneline --left-right HEAD...MERGE_HEAD
< f1270f7 update README
< 9af9d3b add a README
< 694971d update phrase to hola world
> e3eb223 add more tests
> 7cff591 add testing script
> c3ffff1 changed text to hello mundo
```

Esa es una buena lista de los seis compromisos involucrados, así como en qué línea de desarrollo estuvo cada compromiso.

Sin embargo, podemos simplificar aún más esto para darnos un contexto mucho más específico. Si añadimos la opción `--merge` a `git log`, solo mostrará los compromisos en cualquier lado de la fusión que toque un archivo que esté actualmente en conflicto.

```
$ git log --oneline --left-right --merge
< 694971d update phrase to hola world
> c3ffff1 changed text to hello mundo
```

En su lugar, si corremos eso con la opción `-p` obtienes, solo los diffs del archivo que terminó en conflicto. Esto puede ser **bastante**útil al rápidamente darte el contexto que necesitas para ayudarte a entender por qué algo crea problemas y cómo resolverlo de una forma más inteligente.

Formato Diff Combinado

Dado que las etapas de Git clasifican los resultados que tienen éxito, cuando corres `git diff` mientras está en un estado de conflicto de fusión, solo puedes obtener lo que está actualmente en conflicto. Esto puede ser útil para ver lo que todavía debes resolver.

Cuando corres directamente `git diff` después de un conflicto de fusión, te dará la información en un formato de salida diff bastante único.

```
$ git diff
diff --cc hello.rb
index 0399cd5,59727f0..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,11 @@@
  #! /usr/bin/env ruby

  def hello
++<<<<<<< HEAD
 +  puts 'hola world'
++=======
+   puts 'hello mundo'
++>>>>>>> mundo
  end

  hello()
```

El formato es llamado “Diff combinado” y proporciona dos columnas de datos al lado de cada línea. La primera columna muestra si esa línea es diferente (añadido o removido) entre la rama “nuestra” y el archivo en tu directorio de trabajo, y la segunda columna hace lo mismo entre la rama “de ellos” y la copia de tu directorio de trabajo.

Así que en ese ejemplo se puede observar que las líneas <<<<<<< y >>>>>>> están en la copia de trabajo, pero no en ningún lado de la fusión. Esto tiene sentido porque la herramienta de fusión las mantiene ahí para nuestro contexto, pero se espera removamos.

Si resolvemos el conflicto y corremos `git diff` de nuevo, veremos la misma cosa, pero es un poco más útil.

```
$ vim hello.rb
$ git diff
diff --cc hello.rb
index 0399cd5,59727f0..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
  #! /usr/bin/env ruby

  def hello
-   puts 'hola world'
 -  puts 'hello mundo'
++  puts 'hola mundo'
  end

  hello()
```

Esto muestra que “hola mundo” estaba de nuestro lado, pero no en la copia de trabajo, que “hello

mundo” estaba en el lado de ellos, pero no en la copia de trabajo y finalmente que “hola mundo” no estaba en ningún lado, sin embargo está ahora en la copia de trabajo. Esto puede ser útil para revisar antes de comprometer la resolución.

También se puede obtener desde el `git log` para cualquier fusión después del hecho, para ver cómo algo se resolvió luego de este. Git dará salida a este formato si se puede correr `git show` en un compromiso de fusión, o si se añade la opción `--cc` a un `git log -p` (el cual por defecto solo muestras parches para compromisos no fusionados).

```
$ git log --cc -p -1
commit 14f41939956d80b9e17bb8721354c33f8d5b5a79
Merge: f1270f7 e3eb223
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Sep 19 18:14:49 2014 +0200

    Merge branch 'mundo'

    Conflicts:
        hello.rb

diff --cc hello.rb
index 0399cd5,59727f0..e1d0799
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
  #! /usr/bin/env ruby

  def hello
-   puts 'hola world'
 -  puts 'hello mundo'
++  puts 'hola mundo'
  end

  hello()
```

Deshaciendo Fusiones

Ahora que ya conoces como crear un merge commit (compromiso de fusión), probablemente hayas creado algunos por error. Una de las ventajas de trabajar con Git es que está bien cometer errores, porque es posible (y en muchos casos es fácil) solucionarlos.

Los compromisos de fusión no son diferentes. Digamos que comenzaste a trabajar en una rama temática accidentalmente fusionada en una rama `master`, y ahora el historial de compromiso se ve así:

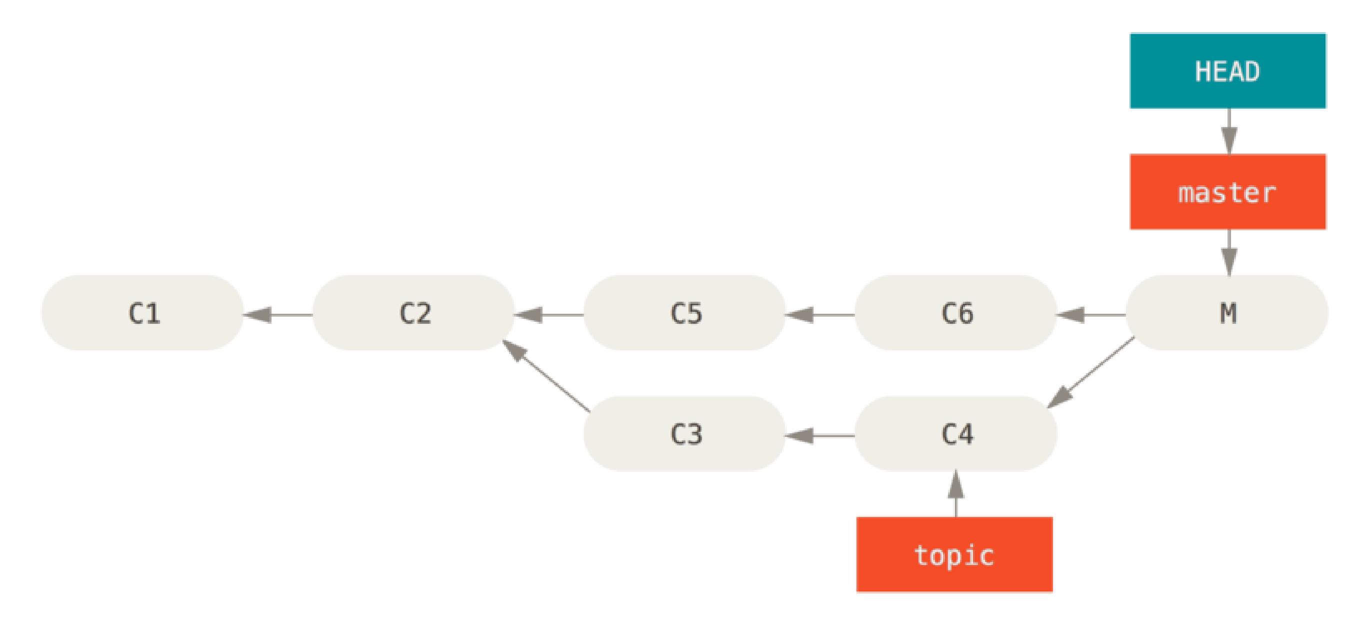


Figure 138. Accidental merge commit

Existen dos formas de abordar este problema, dependiendo de cuál es el resultado que deseas.

Solucionar las referencias

Si el compromiso de fusión no deseado solo existe en tu repositorio local, la mejor y más fácil solución es mover las ramas para que así apunten a dónde quieres que lo hagan. En la mayoría de los casos si sigues al errante `git merge` con `git reset --hard HEAD~`, esto restablecerá los punteros de la rama, así que se verá así:

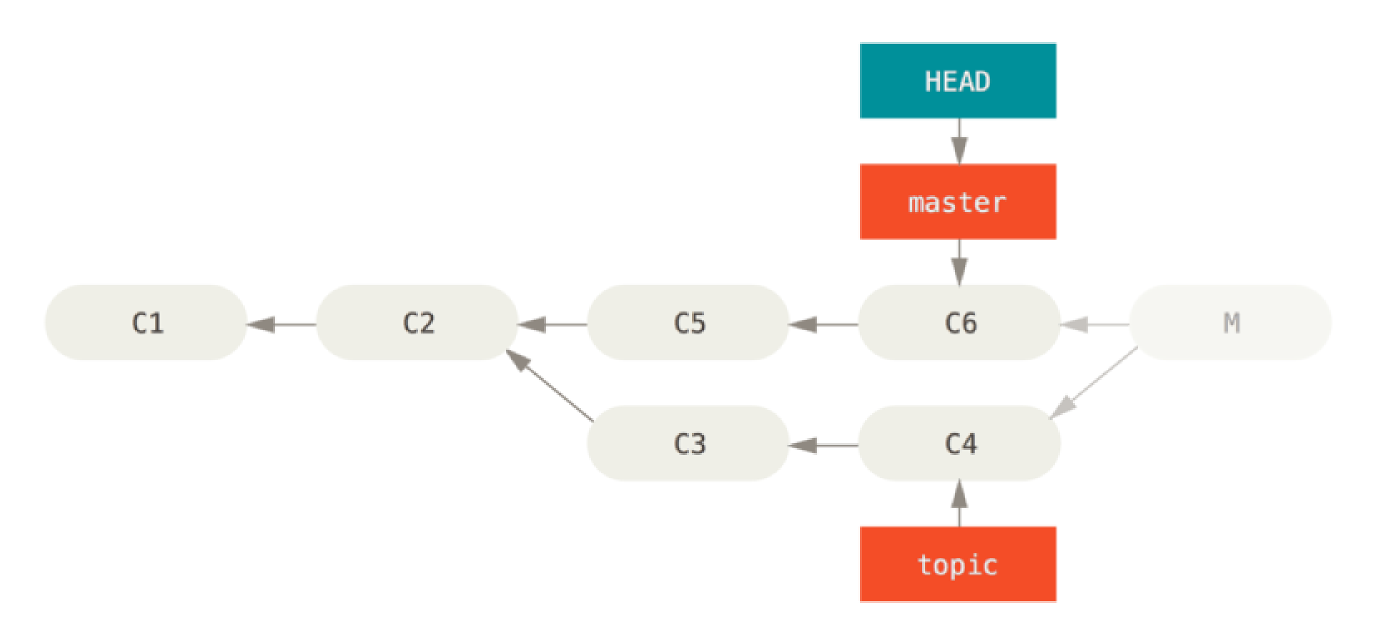


Figure 139. History after `git reset --hard HEAD~`

Ya vimos `restablecer` de nuevo en Reiniciar Desmitificado, así que no debería ser muy difícil averiguar lo que está sucediendo ahí Aquí un repaso rápido: `reset --hard` usualmente va a través de tres pasos:

1. Mover los puntos de la rama HEAD. En este caso, se quiere mover la `principal`a donde se encontraba antes el compromiso de fusión (`C6`).

2. Hacer que el índice parezca HEAD.
3. Hacer que el directorio de trabajo parezca el índice.

La desventaja de este enfoque es que es reescribir el historial, lo cual puede ser problemático con un depósito compartido. Revisa Los Peligros de Reorganizar para más de lo que puede suceder; la versión corta es que si otras personas tienen los compromisos que estas reescribiendo, probablemente deberías evitar `resetear`. Este enfoque tampoco funcionará si cual quiera de los otros compromisos han sido creados desde la fusión; mover los refs efectivamente perdería esos cambios.

Revertir el compromiso

Si mover los punteros de la rama alrededor no funciona para ti, Git te proporciona la opción de hacer un compromiso (commit) nuevo que deshace todos los cambios de uno ya existente. Git llama a esta operación un ‘`revertir’, y en este escenario en particular, has invocado algo así:

```
$ git revert -m 1 HEAD
[master b1d8379] Revert "Merge branch 'topic'"
```

La bandera `-m 1` indica cuál padre es el “mainline” y debería ser mantenido. Cuando se invoque la fusión en el `HEAD` (`git merge topic`), el nuevo compromiso tiene dos padres: el primero es `HEAD` (`C6`), y el segundo es la punta de la rama siendo fusionada en (`C4`). En este caso, se quiere deshacer todos los cambios introducidos por el fusionamiento en el padre #2 (`C4`), mientras manteniendo todo el contenido del padre #1 (`C6`).

El historial con el compromiso revertido se ve así:

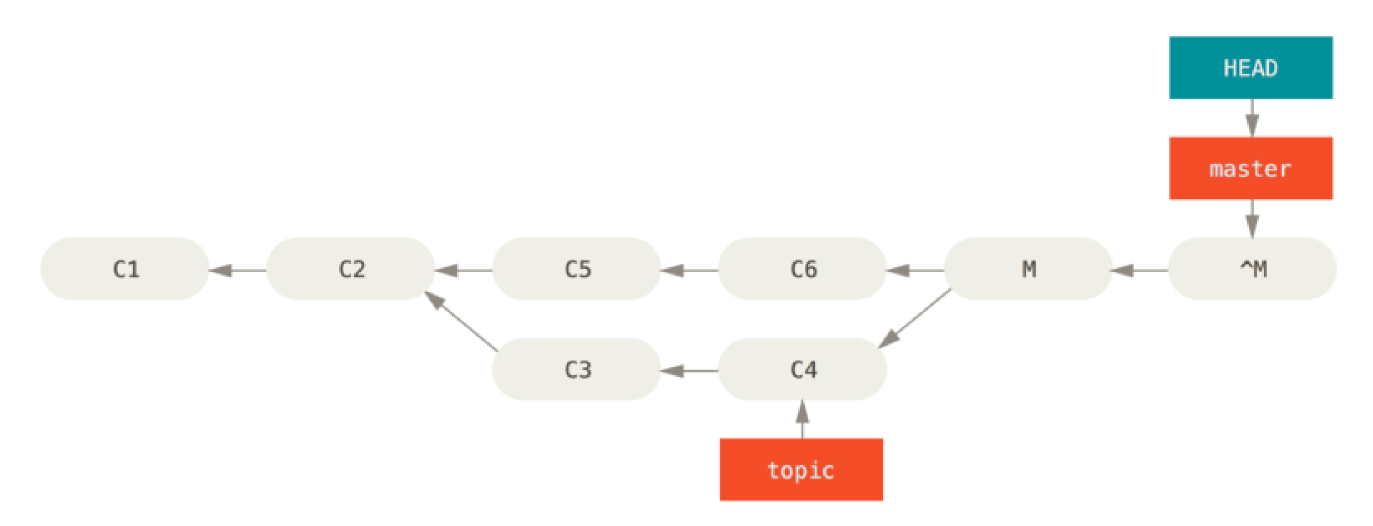


Figure 140. History after `git revert -m 1`

El nuevo compromiso `^M` tiene exactamente los mismos contenidos que `C6`, así que comenzando desde aquí es como si la fusión nunca hubiese sucedido, excepto que la ahora los no fusionados compromisos están todavía en `HEAD`'s history. Git se confundirá si intentas fusionar la rama `temática` en la rama `master`:

```
$ git merge topic
Already up-to-date.
```

No hay nada en topic que no sea ya alcanzable para la master. Que es peor, si añades trabajo a topic y fusionas otra vez, Git solo traerá los cambios desde la fusión revertida:

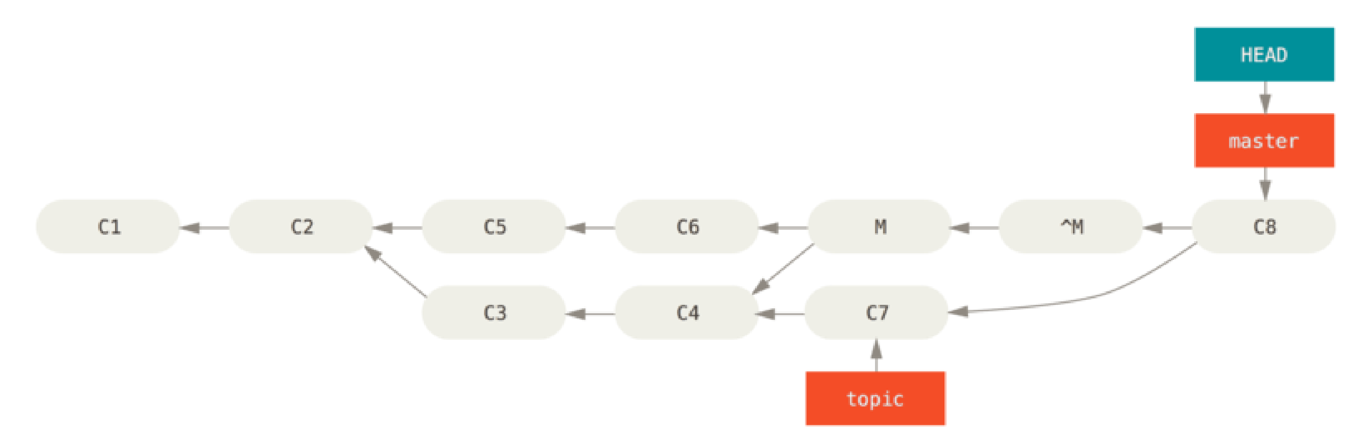


Figure 141. History with a bad merge

La mejor forma de evitar esto es deshacer la fusión original, dado que ahora se quiere traer los cambios que fueron revertidos, **luego** crear un nuevo compromiso de fusión:

```
$ git revert ^M
[master 09f0126] Revert "Revert "Merge branch 'topic'""
$ git merge topic
```

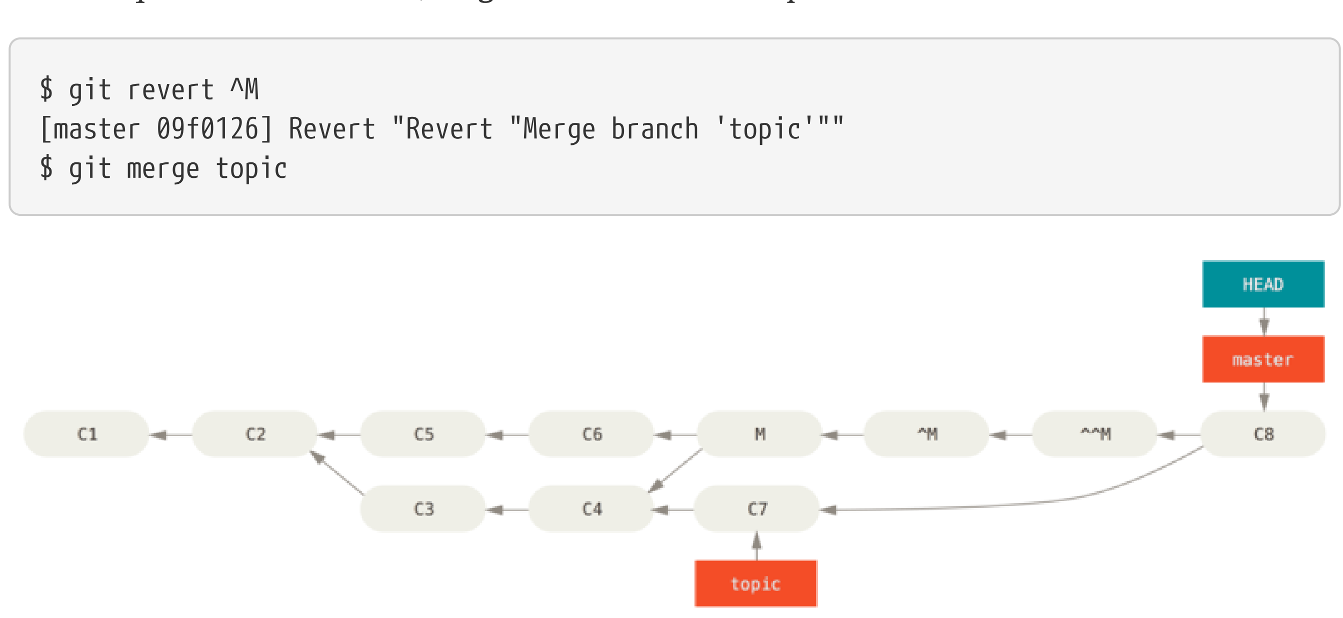


Figure 142. History after re-merging a reverted merge

En este ejemplo, M y ^M se cancelan. Efectivamente ^^M se fusiona en los cambios desde C3 y C4, y C8 se fusiona en los cambios desde C7, así que ahora topic está completamente fusionado.

Otros Tipos de Fusiones

Hasta hora ya cubrimos la fusión normal de dos ramas, normalmente manejado con lo que es llamado la estrategia de fusión “recursive”. Sin embargo, hay otras formas de fusionar a las ramas. Cubriremos algunos de ellos rápidamente.

Nuestra o Su preferencia

Primero que nada, hay otra cosa útil que podemos hacer con el modo de fusión “recursive”. Ya se vio que las opciones `ignore-all-space` y `ignore-space-change` las cuales son pasadas con un `-X` pero también le podemos decir a Git que favorezca un lado u otro cuando observe un conflicto.

Por defecto, cuando Git ve un conflicto entre dos ramas siendo fusionadas, añadirá marcadores de conflictos de fusión a los códigos y marca el archivo como conflictivo y te dejará resolverlo. Si prefieres que Git simplemente escoja un lado especifico e ignore el otro, en lugar de dejarte manualmente fusionar el conflicto, puedes pasar el comando de fusión ya sea un `-Xours` o `-Xtheirs`.

Si Git ve esto, no añadirá marcadores de conflictos. Cualquier diferencia que pueda ser fusionable, se fusionará. Cualquier diferencia que entre en conflicto, él simplemente escogerá el lado que especifiques en su totalidad, incluyendo los archivos binarios.

Si volvemos al ejemplo de “hello world” que estábamos utilizando antes, podemos ver que el fusionamiento en nuestra rama causa conflicto.

```
$ git merge mundo
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Resolved 'hello.rb' using previous resolution.
Automatic merge failed; fix conflicts and then commit the result.
```

Sin embargo, si lo corremos con `-Xours` o `-Xtheirs` no lo causa.

```
$ git merge -Xours mundo
Auto-merging hello.rb
Merge made by the 'recursive' strategy.
 hello.rb | 2 +-
 test.sh  | 2 ++
 2 files changed, 3 insertions(+), 1 deletion(-)
 create mode 100644 test.sh
```

En este caso, en lugar de obtener marcadores de conflicto en el archivo con “hello mundo” en un lado y “hola world” en el otro, simplemente escogerá “hola world”. Sin embargo, todos los cambios no conflictivos en esa rama se fusionaron exitosamente.

Esta opción también puede ser trasmitida al comando `git merge-file` que vimos antes al correr algo como esto `git merge-file --ours` para archivos de fusión individuales.

Si quieres realizar algo así, pero Git no ha intentado siquiera fusionar cambios desde el otro lado, hay una opción más draconiana, la cual es la estrategia de fusión “ours” merge *strategy. Esto es diferente de la opción de fusión recursiva “ours” recursive merge _option.*

Esto básicamente hace una fusión falsa. Registrará un nuevo compromiso de fusión con ambas ramas como padres, pero ni siquiera mirará a la rama que estas fusionando. Simplemente registrará como el resultado de la fusión el código exacto en tu rama actual.

```
$ git merge -s ours mundo
Merge made by the 'ours' strategy.
$ git diff HEAD HEAD~
$
```

Puedes observar que no hay diferencia entre la rama en la que estábamos y el resultado de la fusión.

Esto a menudo puede ser útil para básicamente engañar a Git para que piense que una rama ya ha sido fusionada cuando se hace una fusión más adelante. Por ejemplo, decir que has ramificado una rama de “release” y has hecho un poco de trabajo que querrás fusionar de vuelta en tu rama “master” en algún punto. Mientras tanto, algunos arreglos de fallos en la “master” necesitan ser adaptados en la rama de `release`. Se puede fusionar la rama “bugfix” en la de `release` y también `merge -s ours`, la misma rama en la principal (a pesar de que el arreglo ya se encuentre ahí) Así que, más tarde cuando fusiones la de lanzamiento otra vez, no hay conflictos del bugfix.

Convergencia de Subárbol

La idea de la convergencia de subárboles es que usted tiene dos proyectos, de los cuales uno lleva un subdirectorio del otro y viceversa Cuando especifica una convergencia de subárbol, Git suele ser lo suficientemente inteligente para comprender que uno es un subárbol del otro y convergerá apropiadamente.

Veremos un ejemplo donde se añade un proyecto separado a un proyecto existente y luego se converge el código del segundo dentro de un subdirectorio del primero.

Primero, añadiremos la aplicación Rack a nuestro proyecto. Añadiremos el proyecto Rack como referencia remota en nuestro propio proyecto y luego lo colocaremos en su propio branch:

```
$ git remote add rack_remote https://github.com/rack/rack
$ git fetch rack_remote
warning: no common commits
remote: Counting objects: 3184, done.
remote: Compressing objects: 100% (1465/1465), done.
remote: Total 3184 (delta 1952), reused 2770 (delta 1675)
Receiving objects: 100% (3184/3184), 677.42 KiB | 4 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
From https://github.com/rack/rack
 * [new branch]      build      -> rack_remote/build
 * [new branch]      master     -> rack_remote/master
 * [new branch]      rack-0.4   -> rack_remote/rack-0.4
 * [new branch]      rack-0.9   -> rack_remote/rack-0.9
$ git checkout -b rack_branch rack_remote/master
Branch rack_branch set up to track remote branch refs/remotes/rack_remote/master.
Switched to a new branch "rack_branch"
```

Ahora tenemos la raíz del proyecto Rack en nuestro branch `rack_branch` y nuestro proyecto en el branch `master`. Si verifica uno y luego el otro, puede observar que tienen diferentes raíces de

proyecto:

```
$ ls
AUTHORS         KNOWN-ISSUES   Rakefile      contrib         lib
COPYING         README         bin           example         test
$ git checkout master
Switched to branch "master"
$ ls
README
```

Este concepto es algo extraño. No todos los branchs en su repositorio tendrán que ser branchs del mismo proyecto como tal. No es común, porque rara vez es de ayuda, pero es fácil que los branchs contengan historias completamente diferentes.

En este caso, queremos integrar el proyecto Rack a nuestro proyecto `master` como un subdirectorio. Podemos hacer eso en Git con `git read-tree`. Aprenderá más sobre `read-tree` y sus amigos en Los entresijos internos de Git, pero por ahora sepa que este interpreta el árbol raíz de un branch en su area de staging y directorio de trabajo. Sólo cambiamos de vuelta a su branch `master`, e integramos el branch `rack_branch` al subdirectorio `rack` de nuestro branch `master` de nuestro proyecto principal:

```
$ git read-tree --prefix=rack/ -u rack_branch
```

Cuando hacemos commit, parece que tenemos todos los archivos Rack bajo ese subdirectorio - como si los hubiéramos copiado de un tarball. Lo interesante es que podemos facilmente converger cambios de uno de los branchs al otro. Entonces, si el proyecto Rack se actualiza, podemos atraer cambios río arriba alternando a esa branch e incorporando:

```
$ git checkout rack_branch
$ git pull
```

Luego, podemos converger de vuelta esos cambios a nuestro branch `master`. Para incorporar los cambios y rellenar previamente el mensaje de commit, utilice las opciones `--squash` y `--no-commit`, así como la estrategia de convergencia recursiva de la opción `-Xsubtree`. (La estrategia recursiva está aquí por defecto, pero la incluímos para aclarar.)

```
$ git checkout master
$ git merge --squash -s recursive -Xsubtree=rack --no-commit rack_branch
Squash commit -- not updating HEAD
Automatic merge went well; stopped before committing as requested
```

Todos los cambios del proyeto Rack se convergieron y están listos para ser encomendados localmente. También puede hacer lo opuesto - hacer cambios en el subdirectorio `rack` de su master branch y luego convergerlos a su branch `rack_branch` más adelante para entregarlos a los mantenedores o empujarlos río arriba.

Esto nos deja una manera de tener un flujo de trabajo algo similar al flujo de trabajo de submódulo sin utilizar submódulos (de los cuales hablaremos en Submódulos). Podemos mantener branchs con otros proyectos relacionados en nuestro repositorio y convergerlos tipo subárbol a nuestro proyecto ocasionalmente. Esto es bueno por ciertas razones, por ejemplo todo el códido se encomienda a un único lugar. Sin embargo, tiene el defecto de ser un poco más complejo y facilita el cometer errores al reintegrar cambios o empujar accidentalmente un branch a un repositorio con el que no guarda relación.

Otra particularidad es que para diferenciar entre lo que tiene en su subdirectorio `rack` y el código en su branch `rack_branch` - para ver si necesita convergerlos - no puede usar el comando `diff` normal. En lugar de esto, debe ejecutar `git diff-tree` con el branch que desea comparar a:

```
$ git diff-tree -p rack_branch
```

O, para comparar lo que hay en su subdirectorio `rack` con lo que era el branch `master` en el servidor la última vez que usted hizo fetch, ejecute

```
$ git diff-tree -p rack_remote/master
```

Rerere

La funcionalidad del "git rerere" es una característica oculta. El nombre se refiere a "reuse recorded resolution" y como el nombre lo insinúa, te permite pedirle a Git que recuerde cómo resolviste un conflicto de hunk, así que la próxima vez que vea el mismo conflicto, Git puede resolverlo automáticamente por ti.

Hay una serie de escenarios en los cuales esta funcinalidad podría ser realmente útil. Uno de los ejemplos que es mencionado en el manual es que si te quieres asegurar de que una rama temática longeva se unirá limpiamente, pero no quieres tener un montón de commits de unión por la mitad. Con "rerere" encendido, puedes unir ocacionalmente, resolver los conflictos, y luego revertir la unión. Si haces esto continuamente, entonces la unión final debería ser fácil porque "rerere" puede hacer todo por ti automáticamente.

Esta misma táctica puede ser usada si quieres mantener una rama con rebase, de esta manera no tienes que lidiar con los mismos conflictos de rebase cada vez que lo haces. O si quieres tomar una rama que uniste y arreglar un montón de conflictos y entonces decidir hacer rebase en su lugar - probablemente no tengas que hacer todos los mismos conflictos de nuevo.

Otra situación es cuando unes un montón de ramas temáticas en evolución juntas en una head de pruebas ocacionalmente, como el mismo proyecto Git hace frecuentemente. Si las pruebas fallan, puedes rebobinar las uniones y rehacerlas sin la rama teática que hace fallar a las pruebas sin tener que re-resolver los conflictos de nuevo.

Para activar la funcionalidad "rerere", simplemente tienes que ejecutar este ajuste de configuración:

```
$ git config --global rerere.enabled true
```

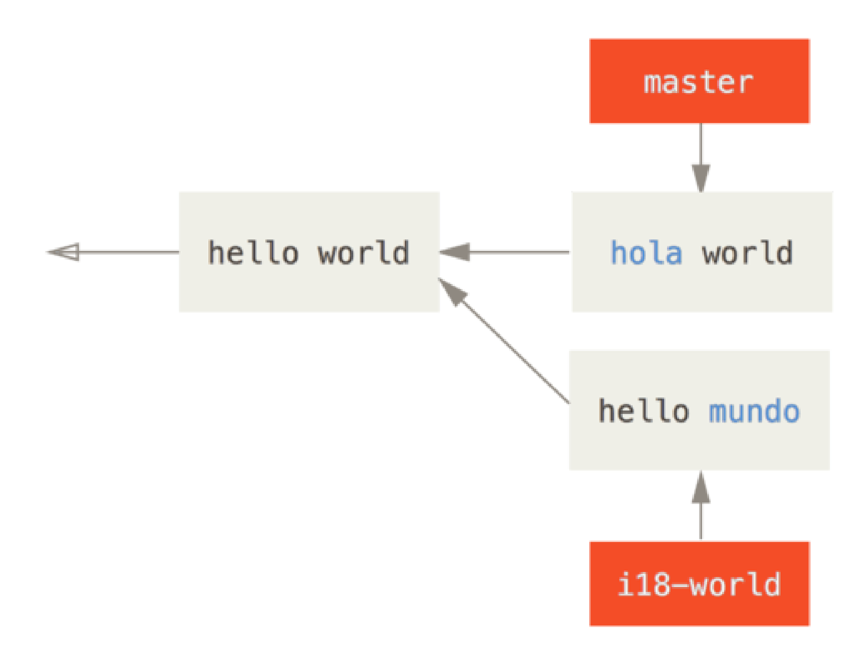
Puedes encenderlo también creando el directorio ".git/rr-cache" en un repositorio específico, pero el ajuste de configuración es limpiador y puede ser hecho globalmente.

Ahora veamos un ejemplo simple, similar al anterior. Digamos que tenemos un archivo que luce de esta manera:

```
#! /usr/bin/env ruby

def hello
  puts 'hello world'
end
```

En una rama, cambiamos la palabra "hello" por "hola", entonces, en otra rama cambiamos el "world" por "mundo", justo como antes.



Cuando unimos las dos ramas juntas, tendremos un conflicto de unión:

```
$ git merge i18n-world
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Recorded preimage for 'hello.rb'
Automatic merge failed; fix conflicts and then commit the result.
```

Deberías notar la nueva línea "Recorded preimage for FILE" ahí adentro. Si no, debería verse exactamente como un conflicto de unión normal. En este punto, "rerere" puede decirnos algunas cosas. Normalmente, podrías ejecutar "git status" en este punto para ver todo lo que entró en conflicto:

```
$ git status
# On branch master
# Unmerged paths:
#   (use "git reset HEAD <file>..." to unstage)
#   (use "git add <file>..." to mark resolution)
#
#   both modified:      hello.rb
#
```

Sin embargo, "git rerere" también te dirá lo que ha registrado el estado pre-unión con "git rerere status":

```
$ git rerere status
hello.rb
```

Y "git rerere diff" mostrará el estado actual de la resolución - con lo que comenzaste a resolver y lo que has resuelto.

```
$ git rerere diff
--- a/hello.rb
+++ b/hello.rb
@@ -1,11 +1,11 @@
 #! /usr/bin/env ruby

 def hello
-<<<<<<<
-  puts 'hello mundo'
-=======
+<<<<<<< HEAD
   puts 'hola world'
->>>>>>>
+=======
+  puts 'hello mundo'
+>>>>>>> i18n-world
 end
```

Además (y esto no está realmente relacionado a "rerere"), puedes usar "ls-files -u" para ver los archivos que están en conflicto y las versiones anterior, izquierda y derecha:

```
$ git ls-files -u
100644 39804c942a9c1f2c03dc7c5ebcd7f3e3a6b97519 1   hello.rb
100644 a440db6e8d1fd76ad438a49025a9ad9ce746f581 2   hello.rb
100644 54336ba847c3758ab604876419607e9443848474 3   hello.rb
```

Ahora puedes resolverlo para ser simplemente `puts 'hola mundo'` y puedes ejecutar el comando "rerere diff" de nuevo para ver lo que "rerere" recordará:

```
$ git rerere diff
--- a/hello.rb
+++ b/hello.rb
@@ -1,11 +1,7 @@
 #! /usr/bin/env ruby

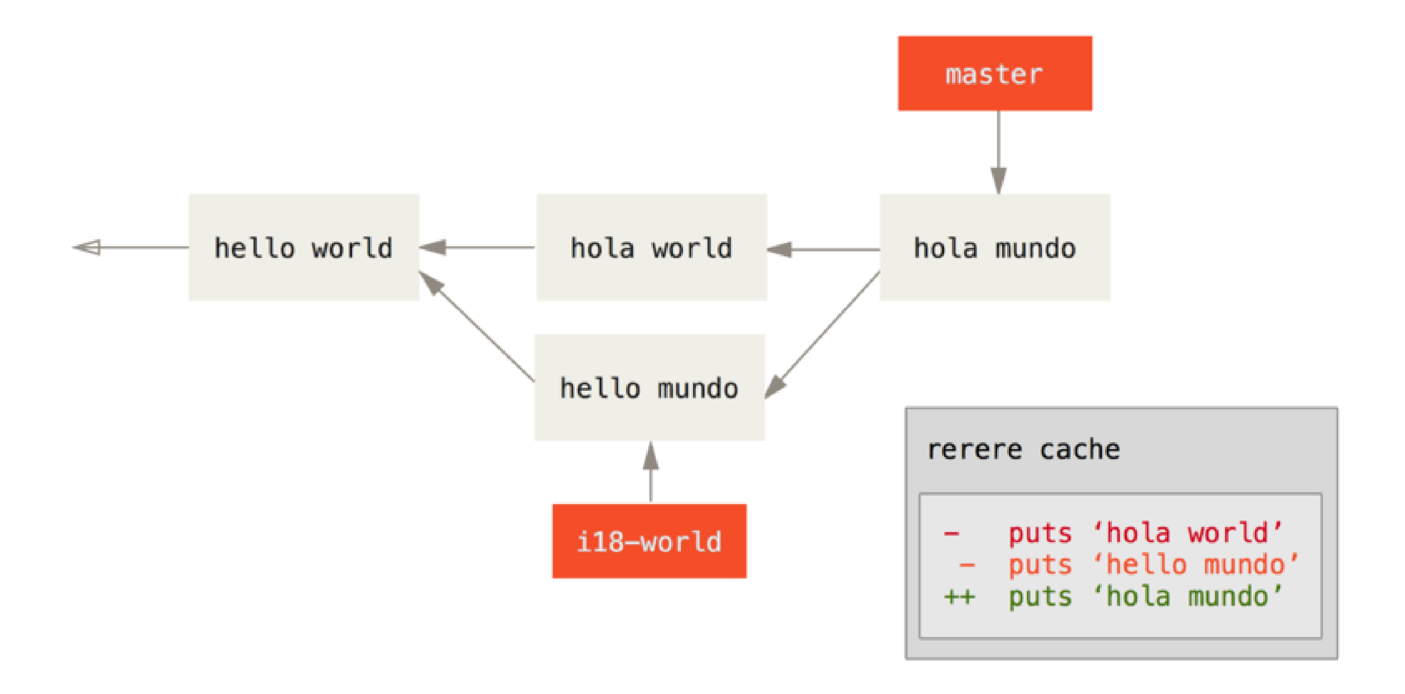
 def hello
-<<<<<<<
-  puts 'hello mundo'
-=======
-  puts 'hola world'
->>>>>>>
+  puts 'hola mundo'
 end
```

Eso básicamente dice, cuando Git ve un conflicto de hunk en un archivo "hello.rb" que tiene "hello mundo" en un lado y "hola world" en el otro, lo resolverá como "hola mundo".

Ahora podemos marcarlo como resuleto y hacerle commit:

```
$ git add hello.rb
$ git commit
Recorded resolution for 'hello.rb'.
[master 68e16e5] Merge branch 'i18n'
```

Ahora podemos ver que "Recorded resolution for FILE" (Registró solución para ARCHIVO).



Ahroa, deshagamos esa unión y luego hagámosle rebase en la cima de nuestra rama maestra en su lugar. Podemos tener nuestra rama de vuelta usando "reset" como vimos en Reiniciar

Desmitificado.

```
$ git reset --hard HEAD^
HEAD is now at ad63f15 i18n the hello
```

Nuestra unión no está hecha. Ahora hagámos rebase a la rama temática.

```
$ git checkout i18n-world
Switched to branch 'i18n-world'

$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: i18n one word
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Resolved 'hello.rb' using previous resolution.
Failed to merge in the changes.
Patch failed at 0001 i18n one word
```

Ahora, tenemos el mismo conflicto de unión que esperábamos, pero échale un vistaso a la línea "Resolved FILE using previous resolution". Si le echamos un vistazo al archivo, veremos que ya está resuleto, ya no hay marcas de conflicto de unión en él.

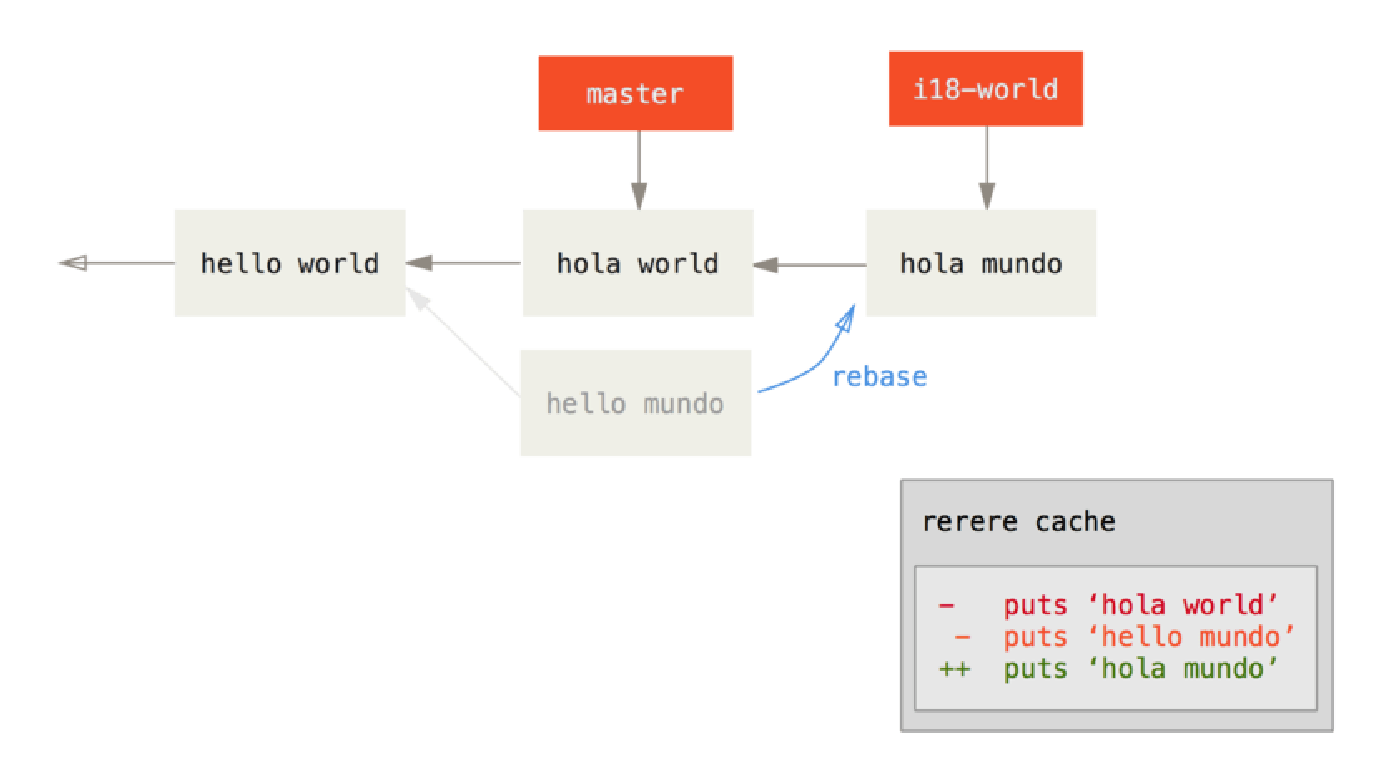
```
$ cat hello.rb
#! /usr/bin/env ruby

def hello
  puts 'hola mundo'
end
```

Además, "git diff" te mostrará cómo fue re-resuleto automáticamente:

```
$ git diff
diff --cc hello.rb
index a440db6,54336ba..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
  #! /usr/bin/env ruby

  def hello
-   puts 'hola world'
 -  puts 'hello mundo'
++  puts 'hola mundo'
  end
```



Puedes también recrear el archivo en conflicto con el comando "checkout":

```
$ git checkout --conflict=merge hello.rb
$ cat hello.rb
#! /usr/bin/env ruby

def hello
<<<<<<< ours
  puts 'hola world'
=======
  puts 'hello mundo'
>>>>>>> theirs
end
```

Vimos un ejemplo de esto en Fusión Avanzada. Por ahora, aunque resolvámoslo sólo ejecuntando

"rerere" de nuevo:

```
$ git rerere
Resolved 'hello.rb' using previous resolution.
$ cat hello.rb
#! /usr/bin/env ruby

def hello
  puts 'hola mundo'
end
```

Hemos re-resuelto el archivo automáticamente usando la resolución en caché "rerere". Ahora puedes añadir y continuar el rebase para completarlo.

```
$ git add hello.rb
$ git rebase --continue
Applying: i18n one word
```

Entonces, si haces muchas re-uniones, o quieres mantener una rama temática actualizada con tu rama maestra sin un montón de uniones, o haces rebase a menudo, puedes encender "rerere" para ayudar un poco a tu vida.

Haciendo debug con Git

Git también provee unas cuantas herramientas para realizar un debug a los problemas en tus proyectos. Porque Git está diseñado para trabajar con casi cualquier tipo de proyecto, estas herramientas son bastante genéricas, pero pueden ayudar a cazar bugs o al culpable cuando las cosas salen mal.

Anotaciones de archivo

Si rastreas un bug en tu código y quieres saber cuándo fue introducido y por qué, la anotación de archivo será muchas veces tu mejor herramienta. Esta te muestra qué commit fue el último en modificar cada línea de cualquier archivo. Así que, si ves que un método en tu código tiene bugs, puedes anotar el archivo con `git blame` para ver cuándo cada línea del método fue editada por última vez y por quién. Este ejemplo usa la opción `-L` para limitar la salida desde las líneas 12 a 22:

```
$ git blame -L 12,22 simplegit.rb
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 12)  def show(tree = 'master')
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 13)   command("git show #{tree}")
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 14)  end
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 15)
9f6560e4 (Scott Chacon  2008-03-17 21:52:20 -0700 16)  def log(tree = 'master')
79eaf55d (Scott Chacon  2008-04-06 10:15:08 -0700 17)   command("git log #{tree}")
9f6560e4 (Scott Chacon  2008-03-17 21:52:20 -0700 18)  end
9f6560e4 (Scott Chacon  2008-03-17 21:52:20 -0700 19)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 20)  def blame(path)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 21)   command("git blame #{path}")
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 22)  end
```

Nota que el primer campo es la parcial de SHA-1 del commit que modificó esa línea. Los siguientes dos campos son valores extraídos del commit- el nombre del autor y la fecha del commit - así podrás ver de manera sencilla quién modificó esa línea y cuándo. Tras estos viene el número de línea y el contenido del archivo. También nota las líneas del commit `^4832fe2`, que designan que esas líneas estuvieron en el commit original del archivo. Ese commit es cuando este archivo fue introducido por primera vez al proyecto, y esas líneas no han sido modificadas desde entonces. Esto es un poco confuso, porque ahora has visto al menos tres maneras diferentes en que Git usa el `^` para modificar el SHA-1 de un commit, pero eso es lo que significa aquí.

Otra cosa interesante de Git es que no rastrea los nombres de archivos de forma explícita. Registra las instantáneas y luego intenta averiguar lo que fue renombrado implícitamente, después del hecho. Una de las características interesantes de esto es que se puede preguntar todo tipo de movimiento de código también. Si pasas `-C` a` git blame`, Git analiza el archivo que estás anotando y trata de averiguar de dónde provienen originalmente fragmentos de código si se copiaron desde otro lugar. Por ejemplo, digamos que estás modificando un archivo llamado `GITServerHandler.m` en múltiples archivos, uno de estos es `GITPackUpload.m`. Realizando un blame a `GITPackUpload.m` con la opción `-C`, se puede ver de dónde vinieron las secciones del código:

```
$ git blame -C -L 141,153 GITPackUpload.m
f344f58d GITServerHandler.m (Scott 2009-01-04 141)
f344f58d GITServerHandler.m (Scott 2009-01-04 142) - (void) gatherObjectShasFromC
f344f58d GITServerHandler.m (Scott 2009-01-04 143) {
70befddd GITServerHandler.m (Scott 2009-03-22 144)         //NSLog(@"GATHER COMMI
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 145)
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 146)         NSString *parentSha;
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 147)         GITCommit *commit = [g
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 148)
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 149)         //NSLog(@"GATHER COMMI
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 150)
56ef2caf GITServerHandler.m (Scott 2009-01-05 151)         if(commit) {
56ef2caf GITServerHandler.m (Scott 2009-01-05 152)                 [refDict setOb
56ef2caf GITServerHandler.m (Scott 2009-01-05 153)
```

Esto es realmente útil. Normalmente, se obtiene como el commit original aquel de dónde se copió el

código, porque esta es la primera vez en la que se tocaron estas líneas en el archivo. Git te informa el commit original donde se escribieron esas líneas, incluso si esto fue hecho en otro archivo.

Búsqueda binaria

Anotar un archivo ayuda si sabe dónde está el problema. Si no sabes lo que está mal, y ha habido decenas o cientos de commits desde el último estado en el que sabes que funcionó el código, probablemente te recurrirás a `git bisect` para obtener ayuda. El comando `bisect` hace una búsqueda binaria a través de su historial de commits para ayudarle a identificar lo más rápidamente posible qué commit introdujo un problema.

Supongamos que acaba de emitir un release de su código en un entorno de producción, está recibiendo informes de errores sobre algo que no estaba ocurriendo en su entorno de desarrollo y no puede imaginar por qué el código lo está haciendo. Regresa a su código, y resulta que puede reproducir el problema, pero no puede averiguar qué está pasando mal. Puede biseccionar el código para averiguarlo. Primero ejecuta `git bisect start` para hacer que las cosas funcionen, y luego usas` git bisect bad` para decirle al sistema que el commit actual está roto. Entonces, debe decir a bisect cuándo fue el último estado bueno conocido, usando `git bisect good [good_commit]`:

```
$ git bisect start
$ git bisect bad
$ git bisect good v1.0
Bisecting: 6 revisions left to test after this
[ecb6e1bc347ccecc5f9350d878ce677feb13d3b2] error handling on repo
```

Git se dio cuenta de que se produjeron alrededor de 12 commits entre el commit que marcó como el último commit bueno (v1.0) y la versión mala actual, y se comprobó el del medio para usted. En este punto, puede ejecutar su prueba para ver si el problema existe a partir de este commit. Si lo hace, entonces se introdujo en algún momento antes de este commit medio; Si no lo hace, entonces el problema se introdujo en algún momento después del commit del medio. Resulta que no hay ningún problema aquí, y le dices a Git escribiendo `git bisect good` y continúa tu viaje:

```
$ git bisect good
Bisecting: 3 revisions left to test after this
[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] secure this thing
```

Ahora estás en otro commit, a medio camino entre el que acabas de probar y tu mala comisión. Ejecuta la prueba de nuevo y descubre que este commit está roto, por lo que le dices a Git que con `git bisect bad`:

```
$ git bisect bad
Bisecting: 1 revisions left to test after this
[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] drop exceptions table
```

Este commit está bien, y ahora Git tiene toda la información que necesita para determinar dónde se introdujo el problema. Le dice que el SHA-1 del primer commit erróneo y muestra algo de la

información del commit y qué archivos se modificaron en ese commit para que pueda averiguar qué sucedió que pueda haber introducido este error:

```
$ git bisect good
b047b02ea83310a70fd603dc8cd7a6cd13d15c04 is first bad commit
commit b047b02ea83310a70fd603dc8cd7a6cd13d15c04
Author: PJ Hyett <pjhyett@example.com>
Date:   Tue Jan 27 14:48:32 2009 -0800

    secure this thing

:040000 040000 40ee3e7821b895e52c1695092db9bdc4c61d1730
f24d3c6ebcfc639b1a3814550e62d60b8e68a8e4 M  config
```

Cuando haya terminado, debe ejecutar `git bisect reset` para reiniciar su HEAD a donde estaba antes de comenzar, o terminará en un estado raro:

```
$ git bisect reset
```

Esta es una herramienta poderosa que puede ayudarle a comprobar cientos de commits para un bug introducido en cuestión de minutos. De hecho, si tiene un script que retornará 0 si el proyecto está bien u otro número si el proyecto está mal, puede automatizar completamente `git bisect`. En primer lugar, vuelva a decirle el alcance de la bisectriz, proporcionando los commits malos y buenos. Puede hacerlo enumerándolos con el comando `bisect start` si lo desea, listando primero el commit malo conocido y el segundo el commit bueno conocido:

```
$ git bisect start HEAD v1.0
$ git bisect run test-error.sh
```

Hacerlo ejecuta automáticamente `test-error.sh` en cada commit de check-out hasta que Git encuentre el primer commit roto. También puede ejecutar algo como `make` o ` make tests` o lo que sea que ejecute pruebas automatizadas para usted.

Submódulos

A menudo ocurre que mientras trabaja en un proyecto, necesita usar otro proyecto desde adentro. Tal vez se trate de una biblioteca desarrollada por un tercero o que ustéd está desarrollando por separado y que se utiliza en múltiples proyectos principales. Un problema común surge en estos escenarios: desea poder tratar los dos proyectos como separados y aún así poder usar uno desde el otro.

Aquí hay un ejemplo. Supongamos que está desarrollando un sitio web y creando feeds Atom. En lugar de escribir su propio código de generación de Atom, decide usar una biblioteca. Es probable que tenga que incluir este código de una biblioteca compartida, como una instalación CPAN o Ruby gem, o copiar el código fuente en su propio árbol de proyectos. El problema con la inclusión de la biblioteca es que es difícil personalizar la biblioteca de alguna manera y, a menudo, es más difícil

de implementar, porque debe asegurarse de que cada cliente tenga esa biblioteca disponible. El problema con el envío del código a su propio proyecto es que cualquier cambio personalizado que realice es difícil de fusionar cuando estén disponibles los cambios de upstream.

Git aborda este problema utilizando submódulos. Los submódulos le permiten mantener un repositorio de Git como un subdirectorio de otro repositorio de Git. Esto le permite clonar otro repositorio en su proyecto y mantener sus commits separados.

Comenzando con los Submódulos

Pasaremos por el desarrollo de un proyecto simple que se ha dividido en un proyecto principal y algunos subproyectos.

Comencemos agregando un repositorio de Git existente como un submódulo del repositorio en el que estamos trabajando. Para agregar un nuevo submódulo, use el comando `git submodule add` con la URL del proyecto que desea empezar a rastrear. En este ejemplo, agregaremos una biblioteca llamada “DbConnector”.

```
$ git submodule add https://github.com/chaconinc/DbConnector
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
```

Por defecto, los submódulos agregarán el subproyecto a un directorio llamado igual que el repositorio, en este caso “DbConnector”. Puede agregar una ruta diferente al final del comando si desea que vaya a otra parte.

Si ejecuta `git status` en este punto, notará algunas cosas.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   .gitmodules
    new file:   DbConnector
```

En primer lugar, debe observar el nuevo archivo `.gitmodules`. Este es un archivo de configuración que almacena la asignación entre la URL del proyecto y el subdirectorio local en el que lo ha insertado:

```
[submodule "DbConnector"]
    path = DbConnector
    url = https://github.com/chaconinc/DbConnector
```

Si tiene múltiples submódulos, tendrá múltiples entradas en este archivo. Es importante tener en cuenta que este archivo está controlado por la versión con sus otros archivos, como su archivo `.gitignore`. Se empuja y hala con el resto de su proyecto. Así es como otras personas que clonan este proyecto saben de dónde obtener los proyectos de submódulos.

NOTE	Dado que la URL en el archivo .gitmodules es lo que otras personas intentarán primero clonar/buscar, asegúrese de usar una URL a la que puedan acceder si es posible. Por ejemplo, si usa una URL diferente a la que presionar para que otros la utilicen, utilice aquella a la que otros tienen acceso. Puede sobrescribir este valor localmente con `git config submodule.DbConnector.url PRIVATE_URL` para su propio uso.

La otra lista en el resultado `git status` es la entrada de la carpeta del proyecto. Si ejecuta `git diff` sobre eso, verá algo interesante:

```
$ git diff --cached DbConnector
diff --git a/DbConnector b/DbConnector
new file mode 160000
index 0000000..c3f01dc
--- /dev/null
+++ b/DbConnector
@@ -0,0 +1 @@
+Subproject commit c3f01dc8862123d317dd46284b05b6892c7b29bc
```

Aunque `DbConnector` es un subdirectorio en su directorio de trabajo, Git lo ve como un submódulo y no rastrea su contenido cuando usted no está en ese directorio. En cambio, Git lo ve como un commit particular de ese repositorio.

Si quieres una mejor salida de diff, puedes pasar la opción `--submodule` a `git diff`.

```
$ git diff --cached --submodule
diff --git a/.gitmodules b/.gitmodules
new file mode 100644
index 0000000..71fc376
--- /dev/null
+++ b/.gitmodules
@@ -0,0 +1,3 @@
+[submodule "DbConnector"]
+       path = DbConnector
+       url = https://github.com/chaconinc/DbConnector
Submodule DbConnector 0000000...c3f01dc (new submodule)
```

Cuando hace commit, ve algo como esto:

```
$ git commit -am 'added DbConnector module'
[master fb9093c] added DbConnector module
 2 files changed, 4 insertions(+)
 create mode 100644 .gitmodules
 create mode 160000 DbConnector
```

Observe el modo `160000` para la entrada` DbConnector`. Ese es un modo especial en Git que básicamente significa que está registrando una confirmación como una entrada de directorio en lugar de un subdirectorio o un archivo.

Clonación de un Proyecto con Submódulos

Aquí clonaremos un proyecto con un submódulo. Cuando clona tal proyecto, de forma predeterminada obtiene los directorios que contienen submódulos, pero ninguno de los archivos dentro de ellos aún:

```
$ git clone https://github.com/chaconinc/MainProject
Cloning into 'MainProject'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 14 (delta 1), reused 13 (delta 0)
Unpacking objects: 100% (14/14), done.
Checking connectivity... done.
$ cd MainProject
$ ls -la
total 16
drwxr-xr-x   9 schacon  staff  306 Sep 17 15:21 .
drwxr-xr-x   7 schacon  staff  238 Sep 17 15:21 ..
drwxr-xr-x  13 schacon  staff  442 Sep 17 15:21 .git
-rw-r--r--   1 schacon  staff   92 Sep 17 15:21 .gitmodules
drwxr-xr-x   2 schacon  staff   68 Sep 17 15:21 DbConnector
-rw-r--r--   1 schacon  staff  756 Sep 17 15:21 Makefile
drwxr-xr-x   3 schacon  staff  102 Sep 17 15:21 includes
drwxr-xr-x   4 schacon  staff  136 Sep 17 15:21 scripts
drwxr-xr-x   4 schacon  staff  136 Sep 17 15:21 src
$ cd DbConnector/
$ ls
$
```

El directorio `DbConnector` está ahí, pero está vacío. Debe ejecutar dos comandos: `git submodule init` para inicializar su archivo de configuración local, y `git submodule update` para buscar todos los datos de ese proyecto y verifique la commit adecuada que figura en su superproyecto:

```
$ git submodule init
Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector) registered for path
'DbConnector'
$ git submodule update
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
Submodule path 'DbConnector': checked out 'c3f01dc8862123d317dd46284b05b6892c7b29bc'
```

Ahora su subdirectorio `DbConnector` está en el estado exacto en el que estaba cuando hizo commit antes.

Sin embargo, hay otra manera de hacer esto que es un poco más simple. Si pasa `--recursive` al comando `git clone`, se inicializará y actualizará automáticamente cada submódulo en el repositorio.

```
$ git clone --recursive https://github.com/chaconinc/MainProject
Cloning into 'MainProject'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 14 (delta 1), reused 13 (delta 0)
Unpacking objects: 100% (14/14), done.
Checking connectivity... done.
Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector) registered for path
'DbConnector'
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
Submodule path 'DbConnector': checked out 'c3f01dc8862123d317dd46284b05b6892c7b29bc'
```

Trabajando en un Proyecto con Submódulos

Ahora tenemos una copia de un proyecto con submódulos y colaboraremos con nuestros compañeros de equipo tanto en el proyecto principal como en el proyecto de submódulo.

Llegada de los Cambios de Upstream

El modelo más simple de usar submódulos en un proyecto sería si simplemente consumiera un subproyecto y quisiera obtener actualizaciones de él de vez en cuando, pero en realidad no estuviera modificando nada en el proceso de pago. Veamos un ejemplo simple allí.

Si desea buscar trabajo nuevo en un submódulo, puede acceder al directorio y ejecutar `git fetch` y

`git merge` en la rama upstream para actualizar el código local.

```
$ git fetch
From https://github.com/chaconinc/DbConnector
   c3f01dc..d0354fc  master     -> origin/master
$ git merge origin/master
Updating c3f01dc..d0354fc
Fast-forward
 scripts/connect.sh | 1 +
 src/db.c           | 1 +
 2 files changed, 2 insertions(+)
```

Ahora, si vuelve al proyecto principal y ejecuta `git diff --submodule` puede ver que el submódulo se actualizó y obtener una lista de commits que se le agregaron. Si no desea escribir `--submodule` cada vez que ejecuta `git diff`, puede establecerlo como el formato predeterminado configurando el valor de configuración `diff.submodule` en "log".

```
$ git config --global diff.submodule log
$ git diff
Submodule DbConnector c3f01dc..d0354fc:
  > more efficient db routine
  > better connection routine
```

Si hace commit en este punto, bloqueará el submódulo para que tenga el nuevo código cuando otras personas lo actualicen.

También hay una forma más sencilla de hacer esto si prefiere no buscar y fusionar manualmente en el subdirectorio. Si ejecuta `git submodule update --remote`, Git irá a sus submódulos y buscará y actualizará por usted.

```
$ git submodule update --remote DbConnector
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
   3f19983..d0354fc  master     -> origin/master
Submodule path 'DbConnector': checked out 'd0354fc054692d3906c85c3af05ddce39a1c0644'
```

Este comando asumirá de forma predeterminada que desea actualizar el pago a la rama `master` del repositorio de submódulos. Sin embargo, puede establecer esto en algo diferente si lo desea. Por ejemplo, si desea que el submódulo DbConnector rastree la rama "stable" del repositorio, puede configurarlo en su archivo `.gitmodules` (para que todos los demás también lo rastreen), o simplemente en su archivo `.git/config` local. Vamos a configurarlo en el archivo `.gitmodules`:

```
$ git config -f .gitmodules submodule.DbConnector.branch stable

$ git submodule update --remote
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
   27cf5d3..c87d55d  stable -> origin/stable
Submodule path 'DbConnector': checked out 'c87d55d4c6d4b05ee34fbc8cb6f7bf4585ae6687'
```

Si deja de lado los `-f .gitmodules`, solo hará el cambio por usted, pero probablemente tenga más sentido rastrear esa información con el repositorio para que todos los demás también lo hagan.

Cuando ejecutamos `git status` en este punto, Git nos mostrará que tenemos “new commits” en el submódulo.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

  modified:   .gitmodules
  modified:   DbConnector (new commits)

no changes added to commit (use "git add" and/or "git commit -a")
```

Si configura `status.submodulesummary`, Git también le mostrará un breve resumen de los cambios a sus submódulos:

```
$ git config status.submodulesummary 1

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

	modified:   .gitmodules
	modified:   DbConnector (new commits)

Submodules changed but not updated:

* DbConnector c3f01dc...c87d55d (4):
  > catch non-null terminated lines
```

En este punto, si ejecuta `git diff` podemos ver que hemos modificado nuestro archivo `.gitmodules` y también que hay un número de commits que hemos eliminado y estamos listos para hacer commit a nuestro proyecto de submódulo.

```
$ git diff
diff --git a/.gitmodules b/.gitmodules
index 6fc0b3d..fd1cc29 100644
--- a/.gitmodules
+++ b/.gitmodules
@@ -1,3 +1,4 @@
[submodule "DbConnector"]
        path = DbConnector
        url = https://github.com/chaconinc/DbConnector
+       branch = stable
 Submodule DbConnector c3f01dc..c87d55d:
  > catch non-null terminated lines
  > more robust error handling
  > more efficient db routine
  > better connection routine
```

Esto es muy bueno ya que podemos ver el registro de las commits a las que estamos a punto de hacer commit en nuestro submódulo. Una vez hecha la commit, puede ver esta información también cuando ejecuta `git log -p`.

```
$ git log -p --submodule
commit 0a24cfc121a8a3c118e0105ae4ae4c00281cf7ae
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Sep 17 16:37:02 2014 +0200

    updating DbConnector for bug fixes

diff --git a/.gitmodules b/.gitmodules
index 6fc0b3d..fd1cc29 100644
--- a/.gitmodules
+++ b/.gitmodules
@@ -1,3 +1,4 @@
 [submodule "DbConnector"]
        path = DbConnector
        url = https://github.com/chaconinc/DbConnector
+       branch = stable
Submodule DbConnector c3f01dc..c87d55d:
  > catch non-null terminated lines
  > more robust error handling
  > more efficient db routine
  > better connection routine
```

Git intentará por defecto actualizar **todos** sus submódulos cuando ejecute `git submodule update --remote` así que si tiene muchos de ellos, quizás quiera pasar el nombre del submódulo que desea intentar actualizar.

Trabajando en un Submódulo

Es bastante probable que si está utilizando submódulos, lo esté haciendo porque realmente desea trabajar en el código dentro del submódulo al mismo tiempo que está trabajando en el código dentro del proyecto principal (o en varios submódulos). . De lo contrario, probablemente usaría un sistema de administración de dependencias más simple (como Maven o Rubygems).

Así que ahora veamos un ejemplo de cómo hacer cambios en el submódulo al mismo tiempo que el proyecto principal y de hacer commit y publicar esos cambios al mismo tiempo.

Hasta ahora, cuando ejecutamos el comando `git submodule update` para obtener cambios de los repositorios de submódulos, Git obtendría los cambios y actualizaría los archivos en el subdirectorio, pero deja el sub-repositorio en lo que se llama un estado “detached HEAD”. Esto significa que no hay una rama de trabajo local (como “master”, por ejemplo) que rastrea los cambios. Por lo tanto, los cambios que realice no serán rastreados correctamente.

Para configurar su submódulo para que sea más fácil entrar y piratear, necesita hacer dos cosas. Necesita ingresar a cada submódulo y verificar una rama para trabajar. Entonces necesita decirle a Git qué hacer si ha hecho cambios y luego `git submodule update --remote` extrae un nuevo trabajo de upstream. Las opciones son que puede combinarlas en su trabajo local, o puede tratar de volver a establecer la base de su trabajo local además de los nuevos cambios.

Primero que todo, vamos a nuestro directorio de submódulos y verifiquemos una rama.

```
$ git checkout stable
Switched to branch 'stable'
```

Probemos con la opción “merge”. Para especificarlo manualmente, solo podemos agregar la opción `--merge` a nuestra llamada `update`. Aquí veremos que hubo un cambio en el servidor para este submódulo y se fusionó.

```
$ git submodule update --remote --merge
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
   c87d55d..92c7337  stable     -> origin/stable
Updating c87d55d..92c7337
Fast-forward
 src/main.c | 1 +
 1 file changed, 1 insertion(+)
Submodule path 'DbConnector': merged in '92c7337b30ef9e0893e758dac2459d07362ab5ea'
```

Si vamos al directorio de DbConnector, tenemos los nuevos cambios ya fusionados en nuestra rama `stable` local. Ahora veamos qué sucede cuando hacemos nuestro propio cambio local a la biblioteca y alguien más impulsa otro cambio en sentido ascendente al mismo tiempo.

```
$ cd DbConnector/
$ vim src/db.c
$ git commit -am 'unicode support'
[stable f906e16] unicode support
 1 file changed, 1 insertion(+)
```

Ahora, si actualizamos nuestro submódulo podemos ver qué sucede cuando hemos realizado un cambio local y upstream también tiene un cambio que necesitamos incorporar.

```
$ git submodule update --remote --rebase
First, rewinding head to replay your work on top of it...
Applying: unicode support
Submodule path 'DbConnector': rebased into '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

Si olvida `--rebase` o `--merge`, Git simplemente actualizará el submódulo a lo que esté en el servidor y restablecerá su proyecto a un estado detached HEAD.

```
$ git submodule update --remote
Submodule path 'DbConnector': checked out '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

Si esto sucede, no se preocupe, simplemente puede volver al directorio y verificar su rama nuevamente (que aún contendrá su trabajo) y combinar o rebasar `origin/stable` (o cualquier rama remota que desee) manualmente.

Si no ha hecho commit a los cambios en su submódulo y ejecuta una actualización de submódulo que podría causar problemas, Git buscará los cambios pero no sobrescribirá el trabajo no guardado en el directorio de su submódulo.

```
$ git submodule update --remote
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 4 (delta 0)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
   5d60ef9..c75e92a  stable     -> origin/stable
error: Your local changes to the following files would be overwritten by checkout:
    scripts/setup.sh
Please, commit your changes or stash them before you can switch branches.
Aborting
Unable to checkout 'c75e92a2b3855c9e5b66f915308390d9db204aca' in submodule path
'DbConnector'
```

Si realizó cambios que entran en conflicto con algo cambiado en upstream, Git le informará cuándo ejecutar la actualización.

```
$ git submodule update --remote --merge
Auto-merging scripts/setup.sh
CONFLICT (content): Merge conflict in scripts/setup.sh
Recorded preimage for 'scripts/setup.sh'
Automatic merge failed; fix conflicts and then commit the result.
Unable to merge 'c75e92a2b3855c9e5b66f915308390d9db204aca' in submodule path
'DbConnector'
```

Puede acceder al directorio de submódulos y solucionar el conflicto tal como lo haría normalmente.

Publicando Cambios de Submódulo

Ahora tenemos algunos cambios en nuestro directorio de submódulos. Algunos de estos fueron enviados desde el inicio por nuestras actualizaciones y otros fueron hechos localmente y todavía no están disponibles para nadie, ya que aún no los hemos promocionado.

```
$ git diff
Submodule DbConnector c87d55d..82d2ad3:
  > Merge from origin/stable
  > updated setup script
  > unicode support
  > remove unnecessary method
  > add new option for conn pooling
```

Si hacemos commit en el proyecto principal y lo elevamos sin elevar los cambios del submódulo también, otras personas que intenten verificar nuestros cambios estarán en problemas ya que no tendrán forma de obtener los cambios del submódulo de los que dependen . Esos cambios solo existirán en nuestra copia local.

Para asegurarse de que esto no ocurra, puede pedirle a Git que verifique que todos sus submódulos se hayan elevado correctamente antes de empujar el proyecto principal. El comando `git push` toma el argumento `--recurse-submodules` que puede configurarse como “check” o “on-demand”. La opción “check” hará que `push` simplemente falle si alguno de los cambios cometidos del submódulo no ha sido elevado.

```
$ git push --recurse-submodules=check
The following submodule paths contain changes that can
not be found on any remote:
  DbConnector

Please try

    git push --recurse-submodules=on-demand

or cd to the path and use

    git push

to push them to a remote.
```

Como puede ver, también nos da algunos consejos útiles sobre lo que podríamos hacer a continuación. La opción simple es ir a cada submódulo y empujar manualmente a los controles remotos para asegurarse de que estén disponibles externamente y luego probar este empuje de nuevo.

La otra opción es usar el valor “on-demand”, que intentará hacer esto por usted.

```
$ git push --recurse-submodules=on-demand
Pushing submodule 'DbConnector'
Counting objects: 9, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (8/8), done.
Writing objects: 100% (9/9), 917 bytes | 0 bytes/s, done.
Total 9 (delta 3), reused 0 (delta 0)
To https://github.com/chaconinc/DbConnector
   c75e92a..82d2ad3  stable -> stable
Counting objects: 2, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 266 bytes | 0 bytes/s, done.
Total 2 (delta 1), reused 0 (delta 0)
To https://github.com/chaconinc/MainProject
   3d6d338..9a377d1  master -> master
```

Como puede ver allí, Git entró en el módulo DbConnector y lo empujó antes de empujar el proyecto principal. Si ese empuje del submódulo falla por algún motivo, el empuje del proyecto principal también fallará.

Fusionando Cambios de Submódulo

Si cambia una referencia de submódulo al mismo tiempo que otra persona, puede tener algunos problemas. Es decir, si los historiales de los submódulos han divergido y les han hecho commit a ramas divergentes en un súper proyecto, puede tomar un poco de trabajo en arreglar.

Si uno de los commits es un antecesor directo del otro (una fusión de avance rápido), entonces Git simplemente elegirá el último para la fusión, por lo que funciona bien.

Sin embargo, Git no intentará siquiera una fusión trivial para ti. Si los commits del submódulo divergen y necesitan fusionarse, obtendrá algo que se ve así:

```
$ git pull
remote: Counting objects: 2, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 2 (delta 1), reused 2 (delta 1)
Unpacking objects: 100% (2/2), done.
From https://github.com/chaconinc/MainProject
   9a377d1..eb974f8  master     -> origin/master
Fetching submodule DbConnector
warning: Failed to merge submodule DbConnector (merge following commits not found)
Auto-merging DbConnector
CONFLICT (submodule): Merge conflict in DbConnector
Automatic merge failed; fix conflicts and then commit the result.
```

Básicamente, lo que sucedió aquí es que Git ha descubierto que las dos ramas registran puntos en la historia del submódulo que son divergentes y necesitan fusionarse. Lo explica como “merge

following commits not found", lo cual es confuso pero explicaremos el por qué en un momento.

Para resolver el problema, debe averiguar en qué estado debería estar el submódulo. Curiosamente, Git no le da demasiada información para ayudar aquí, ni siquiera los SHA-1 de las commits de ambos lados de la historia. Afortunadamente, es fácil de entender. Si ejecuta `git diff`, puede obtener los SHA-1 de las commits registradas en ambas ramas que estaba intentando fusionar.

```
$ git diff
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
```

Entonces, en este caso, `eb41d76` es la commit en nuestro submódulo que **nosotros** teníamos y `c771610` es la commit que tenía upstream. Si vamos a nuestro directorio de submódulos, ya debería estar en `eb41d76` ya que la fusión no lo habría tocado. Si por alguna razón no es así, simplemente puede crear y verificar una rama que lo señale.

Lo que es importante es el SHA-1 de la commit del otro lado. Esto es lo que tendrás que fusionar y resolver. Puedes simplemente probar la fusión con el SHA-1 directamente, o puedes crear una rama para él y luego intentar fusionar eso. Sugeriríamos este último, aunque sea para hacer un mejor mensaje de commit de fusión.

Por lo tanto, accederemos a nuestro directorio de submódulos, crearemos una rama basada en ese segundo SHA-1 de `git diff` y fusionaremos manualmente.

```
$ cd DbConnector

$ git rev-parse HEAD
eb41d764bccf88be77aced643c13a7fa86714135

$ git branch try-merge c771610
(DbConnector) $ git merge try-merge
Auto-merging src/main.c
CONFLICT (content): Merge conflict in src/main.c
Recorded preimage for 'src/main.c'
Automatic merge failed; fix conflicts and then commit the result.
```

Aquí tenemos un conflicto de fusión real, por lo que si resolvemos eso y le hacemos commit, podemos simplemente actualizar el proyecto principal con el resultado.

```
$ vim src/main.c ①
$ git add src/main.c
$ git commit -am 'merged our changes'
Recorded resolution for 'src/main.c'.
[master 9fd905e] merged our changes

$ cd .. ②
$ git diff ③
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
@@@ -1,1 -1,1 +1,1 @@@
- Subproject commit eb41d764bccf88be77aced643c13a7fa86714135
 -Subproject commit c77161012afbbe1f58b5053316ead08f4b7e6d1d
++Subproject commit 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a
$ git add DbConnector ④

$ git commit -m "Merge Tom's Changes" ⑤
[master 10d2c60] Merge Tom's Changes
```

① Primero resolvemos el conflicto

② Luego volvemos al directorio principal del proyecto

③ Podemos verificar los SHA-1 nuevamente

④ Resolver la entrada conflictiva del submódulo

⑤ Commit a nuestra fusión

Puede ser un poco confuso, pero realmente no es muy difícil.

Curiosamente, hay otro caso que maneja Git. Si existe un commit de fusión en el directorio del submódulo que contiene **ambos** commits en su historial, Git lo sugerirá como posible solución. Se ve que en algún punto del proyecto del submódulo, alguien fusionó las ramas que contienen estos dos commits, así que tal vez querrá esa.

Esta es la razón por la cual el mensaje de error de antes era “fusionar los siguientes commits no encontrado”, porque no podía hacer **esto**. Es confuso porque ¿quién esperaría que **intentara** hacer esto?

Si encuentra una única commit de fusión aceptable, verá algo como esto:

```
$ git merge origin/master
warning: Failed to merge submodule DbConnector (not fast-forward)
Found a possible merge resolution for the submodule:
 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a: > merged our changes
If this is correct simply add it to the index for example
by using:

  git update-index --cacheinfo 160000 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a
"DbConnector"

which will accept this suggestion.
Auto-merging DbConnector
CONFLICT (submodule): Merge conflict in DbConnector
Automatic merge failed; fix conflicts and then commit the result.
```

Lo que está sugiriendo que haga es actualizar el índice como si hubiera ejecutado `git add`, que borra el conflicto y luego haga commit. Sin embargo, probablemente no debería hacer esto. También puede acceder fácilmente al directorio de submódulos, ver cuál es la diferencia, avanzar rápidamente a esta commit, probarla correctamente y luego hacerle commit.

```
$ cd DbConnector/
$ git merge 9fd905e
Updating eb41d76..9fd905e
Fast-forward

$ cd ..
$ git add DbConnector
$ git commit -am 'Fast forwarded to a common submodule child'
```

Esto logra lo mismo, pero al menos de esta manera puede verificar que funcione y que tenga el código en el directorio de su submódulo cuando haya terminado.

Consejos de Cubmódulo

Hay algunas cosas que puede hacer para facilitar el trabajo con los submódulos.

Submódulo Foreach

Hay un comando del submódulo `foreach` para ejecutar algún comando arbitrario en cada submódulo. Esto puede ser realmente útil si tiene un número de submódulos en el mismo proyecto.

Por ejemplo, digamos que queremos comenzar una nueva característica o hacer una corrección de errores y tenemos trabajo sucediendo en varios submódulos. Podemos esconder fácilmente todo el trabajo en todos nuestros submódulos.

```
$ git submodule foreach 'git stash'
Entering 'CryptoLibrary'
No local changes to save
Entering 'DbConnector'
Saved working directory and index state WIP on stable: 82d2ad3 Merge from
origin/stable
HEAD is now at 82d2ad3 Merge from origin/stable
```

Entonces podemos crear una nueva rama y cambiar a ella en todos nuestros submódulos.

```
$ git submodule foreach 'git checkout -b featureA'
Entering 'CryptoLibrary'
Switched to a new branch 'featureA'
Entering 'DbConnector'
Switched to a new branch 'featureA'
```

Entiende la idea. Una cosa realmente útil que puede hacer es producir una buena diff unificada de lo que ha cambiado en su proyecto principal y todos sus subproyectos también.

```
$ git diff; git submodule foreach 'git diff'
Submodule DbConnector contains modified content
diff --git a/src/main.c b/src/main.c
index 210f1ae..1f0acdc 100644
--- a/src/main.c
+++ b/src/main.c
@@ -245,6 +245,8 @@ static int handle_alias(int *argcp, const char ***argv)

      commit_pager_choice();

+     url = url_decode(url_orig);
+
      /* build alias_argv */
      alias_argv = xmalloc(sizeof(*alias_argv) * (argc + 1));
      alias_argv[0] = alias_string + 1;
Entering 'DbConnector'
diff --git a/src/db.c b/src/db.c
index 1aaefb6..5297645 100644
--- a/src/db.c
+++ b/src/db.c
@@ -93,6 +93,11 @@ char *url_decode_mem(const char *url, int len)
        return url_decode_internal(&url, len, NULL, &out, 0);
 }

+char *url_decode(const char *url)
+{
+       return url_decode_mem(url, strlen(url));
+}
+
 char *url_decode_parameter_name(const char **query)
 {
        struct strbuf out = STRBUF_INIT;
```

Aquí podemos ver que estamos definiendo una función en un submódulo y llamándola en el proyecto principal. Obviamente, este es un ejemplo simplificado, pero con suerte le da una idea de cómo esto puede ser útil.

Alias Útiles

Es posible que desee configurar algunos alias para algunos de estos comandos, ya que pueden ser bastante largos y no puede establecer opciones de configuración para la mayoría de ellos para que sean predeterminados. Cubrimos la configuración de alias de Git en Alias de Git, pero aquí hay un ejemplo de lo que puede querer configurar si planea trabajar mucho con submódulos en Git.

```
$ git config alias.sdiff '!'"git diff && git submodule foreach 'git diff'"
$ git config alias.spush 'push --recurse-submodules=on-demand'
$ git config alias.supdate 'submodule update --remote --merge'
```

De esta forma, simplemente puede ejecutar `git update` cuando desee actualizar sus submódulos, o `git push` para presionar con la comprobación de dependencia de submódulos.

Problemas con los Submódulos

Sin embargo, el uso de submódulos no deja de tener problemas.

Por ejemplo, cambiar ramas con submódulos en ellos también puede ser complicado. Si crea una nueva rama, agrega un submódulo allí y luego vuelve a una rama sin ese submódulo, aún tiene el directorio de submódulo como un directorio sin seguimiento:

```
$ git checkout -b add-crypto
Switched to a new branch 'add-crypto'

$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into 'CryptoLibrary'...
...

$ git commit -am 'adding crypto library'
[add-crypto 4445836] adding crypto library
 2 files changed, 4 insertions(+)
 create mode 160000 CryptoLibrary

$ git checkout master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    CryptoLibrary/

nothing added to commit but untracked files present (use "git add" to track)
```

Eliminar el directorio no es difícil, pero puede ser un poco confuso tener eso allí. Si lo quita y luego vuelve a la rama que tiene ese submódulo, necesitará ejecutar `submodule update --init` para repoblarlo.

```
$ git clean -fdx
Removing CryptoLibrary/

$ git checkout add-crypto
Switched to branch 'add-crypto'

$ ls CryptoLibrary/

$ git submodule update --init
Submodule path 'CryptoLibrary': checked out 'b8dda6aa182ea4464f3f3264b11e0268545172af'

$ ls CryptoLibrary/
Makefile    includes    scripts     src
```

De nuevo, no es realmente muy difícil, pero puede ser un poco confuso.

La otra advertencia principal con la que se topan muchas personas es pasar de subdirectorios a submódulos. Si ha estado rastreando archivos en su proyecto y desea moverlos a un submódulo, debe tener cuidado o Git se enojará con usted. Supongamos que tiene archivos en un subdirectorio de su proyecto y desea cambiarlo a un submódulo. Si elimina el subdirectorio y luego ejecuta `submodule add`, Git le grita:

```
$ rm -Rf CryptoLibrary/
$ git submodule add https://github.com/chaconinc/CryptoLibrary
'CryptoLibrary' already exists in the index
```

Primero debe abandonar el directorio `CryptoLibrary`. Luego puede agregar el submódulo:

```
$ git rm -r CryptoLibrary
$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into 'CryptoLibrary'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
```

Ahora supongamos que hizo eso en una rama. Si intenta volver a una rama donde esos archivos todavía están en el árbol en lugar de un submódulo – obtiene este error:

```
$ git checkout master
error: The following untracked working tree files would be overwritten by checkout:
  CryptoLibrary/Makefile
  CryptoLibrary/includes/crypto.h
  ...
Please move or remove them before you can switch branches.
Aborting
```

Puede forzarlo a cambiar con `checkout -f`, pero tenga cuidado de no tener cambios no guardados allí ya que podrían sobrescribirse con ese comando.

```
$ git checkout -f master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
```

Luego, cuando vuelve, obtiene un directorio `CryptoLibrary` vacío por alguna razón y `git submodule update` tampoco puede arreglarlo. Es posible que deba acceder al directorio de su submódulo y ejecutar un `git checkout .` para recuperar todos sus archivos. Puede ejecutar esto en un script `submodule foreach` para ejecutarlo en múltiples submódulos.

Es importante tener en cuenta que los submódulos de estos días mantienen todos sus datos de Git en el directorio `.git` del proyecto superior, por lo que a diferencia de muchas versiones anteriores de Git, la destrucción de un directorio de submódulos no perderá ningunas ramas o commits que tengas.

Con estas herramientas, los submódulos pueden ser un método bastante simple y efectivo para desarrollar en varios proyectos relacionados pero aún separados simultáneamente.

Agrupaciones

Aunque ya hemos considerado las maneras mas comunes de transferir la base de datos del Git en el internet (HTTP, SSH, etc.), existe aún una manera de hacerlo, aunque no es muy comúnmente usada puede ser muy útil.

El Git es capaz de “agrupar” la base de datos en un único archivo. Esto puede ser muy útil en varios escenarios. Tal vez tu red está caída y quieres enviar cambios a tu co-trabajdores. Quizas estás trabajando en algún lugar sin conexión y no tienes acceso a la red local por motivos de seguridad. Tal vez su tarjeta inalámbrica / ethernet se rompió. Tal vez no tienes acceso al servidor compartido por el momento, y quieres enviar un correo con actualizaciones y no quieres transferir 40 confirmaciones via `format-patch`.

Aquí es donde el comando de `git bundle` es muy útil. El comando `bundle` juntará todo lo que normalmente se empujaría sobre el cable con un comando de` push git ‘en un archivo binario que puedes enviar por correo a alguien o poner un flash drive, luego desglosarlo en otro repositorio.

Veamos un ejemplo simple. Digamos que tienes un repositorio con dos confirmaciones:

```
$ git log
commit 9a466c572fe88b195efd356c3f2bbeccdb504102
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Mar 10 07:34:10 2010 -0800

    Segunda Confirmacion

commit b1ec3248f39900d2a406049d762aa68e9641be25
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Mar 10 07:34:01 2010 -0800

    Primera confirmaciión
```

Si quieres enviar ese repositorio a alguien y no tienes acceso a algún repositorio para hacerlo, o simplemente no quieres configurar uno, puedes juntarlo con el `git bundle create `.

```
$ git bundle create repo.bundle HEAD master
Counting objects: 6, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (6/6), 441 bytes, done.
Total 6 (delta 0), reused 0 (delta 0)
```

Ahora tienes un archive nombrado `repo.bundle` este tiene todos los datos nuevos necesarios para re-crear el repositorio de la rama `maester`. Con el comando de `bundle` necesitaras enlistar cualquier referencia o rango especifico de confirmaciones que quieres que sean incluidas. Si tiene la intención de clonar esto en otro lugar, debe agregar HEAD como referencia, así como lo hemos hecho aquí.

Puedes enviar por correo este archivo `repo.bundle` a alguien más, o ponerlo en una memoria USB y simplemente irte.

Por otro lado, supongamos que se envía este archivo de `repo.bundle` y deseas trabajar en el proyecto. Puedes clonarlo desde el archivo binario en un directorio, como lo harías desde una URL.

```
$ git clone repo.bundle repo
Initialized empty Git repository in /private/tmp/bundle/repo/.git/
$ cd repo
$ git log --oneline
9a466c5 second commit
b1ec324 first commit
```

Si no quieres incluir HEAD en las referencias, tambien tendrás que especificar `-b master` o cualquier rama que sea incluída porque de otra manera no sabrá que rama revisar.

Digamos que ahora haces tres confirmaciones en ello y quieres enviar nuevas confirmaciones via agrupacion en una USB o por correo.

```
$ git log --oneline
71b84da last commit - second repo
c99cf5b fourth commit - second repo
7011d3d third commit - second repo
9a466c5 second commit
b1ec324 first commit
```

Primero necesitamos determinar el rango de confirmaciones que queremos incluir en la agrupación. No como los protocolos de la red que figuraran en el mínimo de datos para transferir desde la red por nosotros, tendremos que hacer esto manualmente. Ahora simplemente puedes hacer la misma cosa y agrupar el repositorio entero, el que trabajará, pero es mejor solo agrupar las diferencias – solamente las tres confirmaciones que hicimos localmente.

Para hacer eso, tienes que calcular la diferencia. Como hemos descrito en Rangos de Commits, Puedes especificar el rango de confirmaciones en un numero de caminos. Para obtener las tres confirmaciones que tenemos en nuestra rama maestra que no estaban en la rama que copiamos originalmente, podemos usar algo como `origin/master..master` o `master ^origin/master`. Puedes probar esto con el comando de `log`.

```
$ git log --oneline master ^origin/master
71b84da last commit - second repo
c99cf5b fourth commit - second repo
7011d3d third commit - second repo
```

Entonces ahora que tenemos la lista de confirmaciones queremos incluirlas en la agrupación, agrupémoslas todas. hacemos eso con el comando de `git bundle create`, dandole un nombre al archivo que queremos que sea parte de la agrupación y el rango de confirmaciones que queremos incluir en el mismo.

```
$ git bundle create commits.bundle master ^9a466c5
Counting objects: 11, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (9/9), 775 bytes, done.
Total 9 (delta 0), reused 0 (delta 0)
```

Ahora tenemos un archivo de `commits.bundle` en nuestro directorio. Si tomamos este y se lo enviamos a nuestro compañero, ella pueda importarlo en el repositorio original, incluso si se ha incluido más trabajo en el tiempo que ha pasado.

Cuando obtenga la agrupación, puede inspeccionarlo para ver que contiene antes de que lo importe en el repositorio. El primer comando es el `bundle verify` que hará saber si el archivo es actualmente una agrupacion válida del Git y asi tendras todos los requerimientos necesarios para reconstituiirlo apropiadamente.

```
$ git bundle verify ../commits.bundle
The bundle contains 1 ref
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
The bundle requires these 1 ref
9a466c572fe88b195efd356c3f2bbeccdb504102 second commit
../commits.bundle is okay
```

Si el agrupador a creado una agrupación de solo las dos últimas confirmaciones que se han hecho, en lugar de las tres, tel repositorio original no será capaz de improtarlo, dado que falta un requisito en la historia. El comando de `verify` debería de verse entonces así:

```
$ git bundle verify ../commits-bad.bundle
error: Repository lacks these prerequisite commits:
error: 7011d3d8fc200abe0ad561c011c3852a4b7bbe95 third commit - second repo
```

Como sea, nuestra primera agrupación es válida, Entonces podemos obtener confirmaciones de ella. si quieres ver que ramas están en la agrupación que pueden ser importadas, de igual manera hay un comando para verlo:

```
$ git bundle list-heads ../commits.bundle
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
```

el sub-comando de `verify` te dira el encabezado tambien. el punto es ver que puede ser puesto, para que puedas usar el comando de fetch `fetch` o `pull` para importar confirmaciones de este archivo. Aquí vamos a buscar la rama *master* de la agrupación por una llamada *other-master* en nuestro repositorio:

```
$ git fetch ../commits.bundle master:other-master
From ../commits.bundle
 * [new branch]      master     -> other-master
```

Ahora podemos ver que hemos importado las confirmaciones a la rama de *other-master'asi mismo como tantas confirmaciones hayamos hecho mientras tanto en nuestra rama 'master* .

```
$ git log --oneline --decorate --graph --all
* 8255d41 (HEAD, master) third commit - first repo
| * 71b84da (other-master) last commit - second repo
| * c99cf5b fourth commit - second repo
| * 7011d3d third commit - second repo
|/
* 9a466c5 second commit
* b1ec324 first commit
```

Entonces, `git bundle` puede ser muy util para compartir o hacer operaciones tipo red cuando no

tienes la red adecuada o repositorio compartido para hacerlo.

Replace

Los objetos de Git son inmutables, pero proporciona una manera interesante de pretender reemplazar objetos en su base de datos con otros objetos.

El comando `replace` le permite especificar un objeto en Git y decir" cada vez que vea esto, fingir que es esta otra cosa ". Esto es más útil para reemplazar un commit en tu historia con otro.

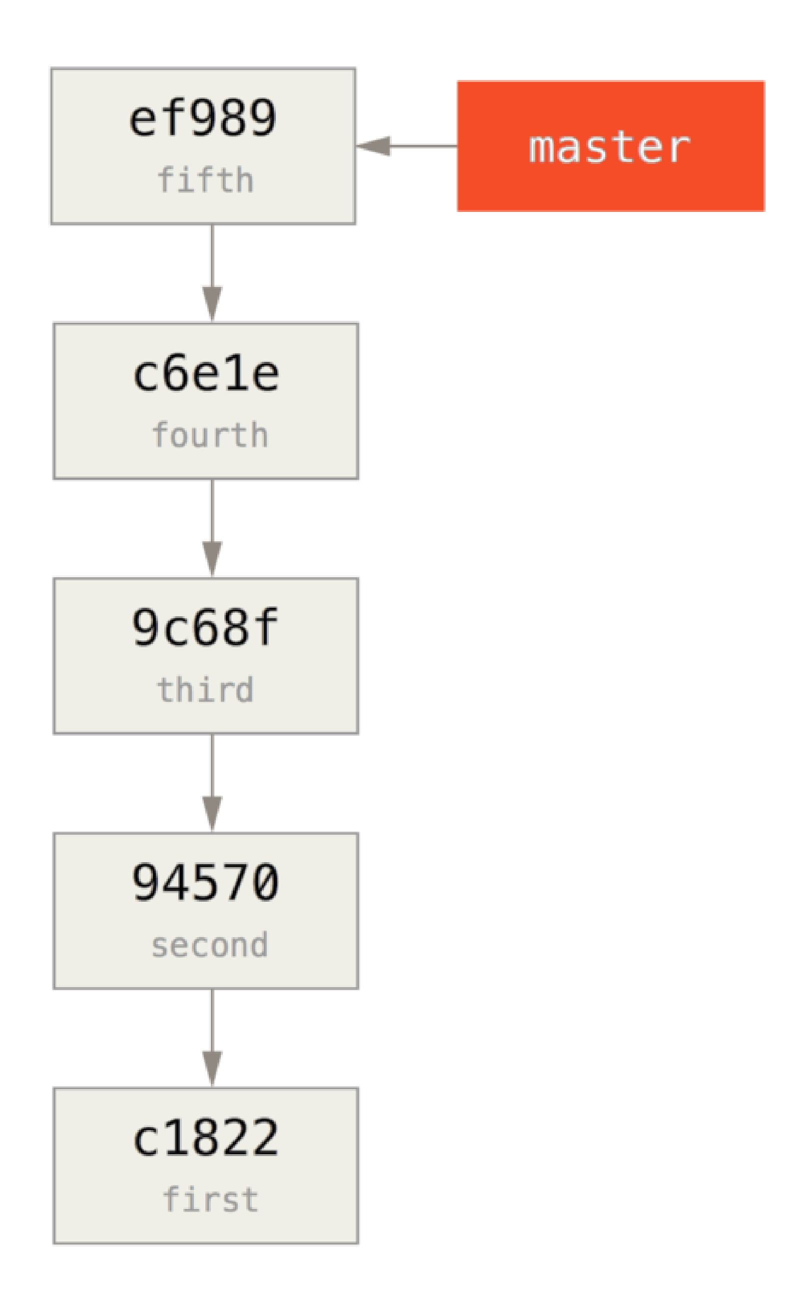
Por ejemplo, supongamos que tiene un gran historial de códigos y desea dividir su repositorio en un breve historial para nuevos desarrolladores y una historia mucho más larga para las personas interesadas en la minería de datos. Puede injertar una historia en la otra mediante `replace`ing el commit más antiguo en la nueva línea con el último commit en el anterior. Esto es bueno porque significa que en realidad no tienes que reescribir cada commit en la nueva historia, como normalmente tendrías que hacer para unirlos juntos (porque el parentesco efectúa los SHA-1s).

Vamos a probar esto. Tomemos un repositorio existente, lo dividimos en dos repositorios, uno reciente y otro histórico, y luego veremos cómo podemos recombinarlos sin modificar los repositorios recientes SHA-1 a través de `replace`.

Usaremos un repositorio sencillo con cinco compromisos simples:

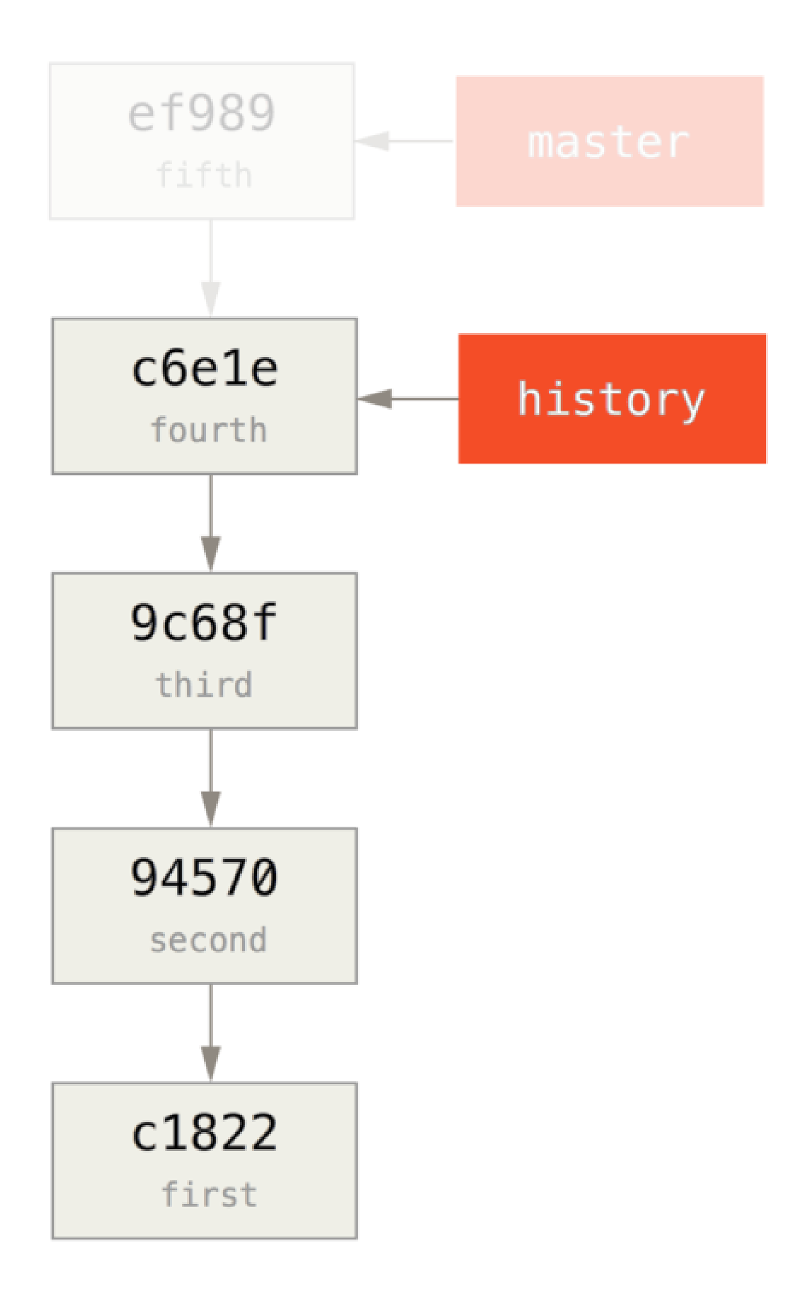
```
$ git log --oneline
ef989d8 fifth commit
c6e1e95 fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

Queremos dividir esto en dos líneas de la historia. Una línea pasa de comprometer uno a cometer cuatro - que será el histórico. La segunda línea sólo se compromete cuatro y cinco - que será la historia reciente.



Bueno, la creación de la historia histórica es fácil, sólo podemos poner una rama en la historia y luego empujar esa rama a la rama principal de un nuevo repositorio remoto.

```
$ git branch history c6e1e95
$ git log --oneline --decorate
ef989d8 (HEAD, master) fifth commit
c6e1e95 (history) fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```



Ahora podemos hacer push a la nueva rama `history` a la rama` master` de nuestro nuevo repositorio:

```
$ git remote add project-history https://github.com/schacon/project-history
$ git push project-history history:master
Counting objects: 12, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (12/12), 907 bytes, done.
Total 12 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (12/12), done.
To git@github.com:schacon/project-history.git
 * [new branch]      history -> master
```

OK, así que nuestra historia se publica. Ahora la parte más difícil es truncar nuestra historia reciente por lo que es más pequeño. Necesitamos una superposición para que podamos reemplazar un commit en uno con un commit equivalente en el otro, por lo que vamos a truncar esto a sólo comete cuatro y cinco (así cometer cuatro superposiciones).

```
$ git log --oneline --decorate
ef989d8 (HEAD, master) fifth commit
c6e1e95 (history) fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

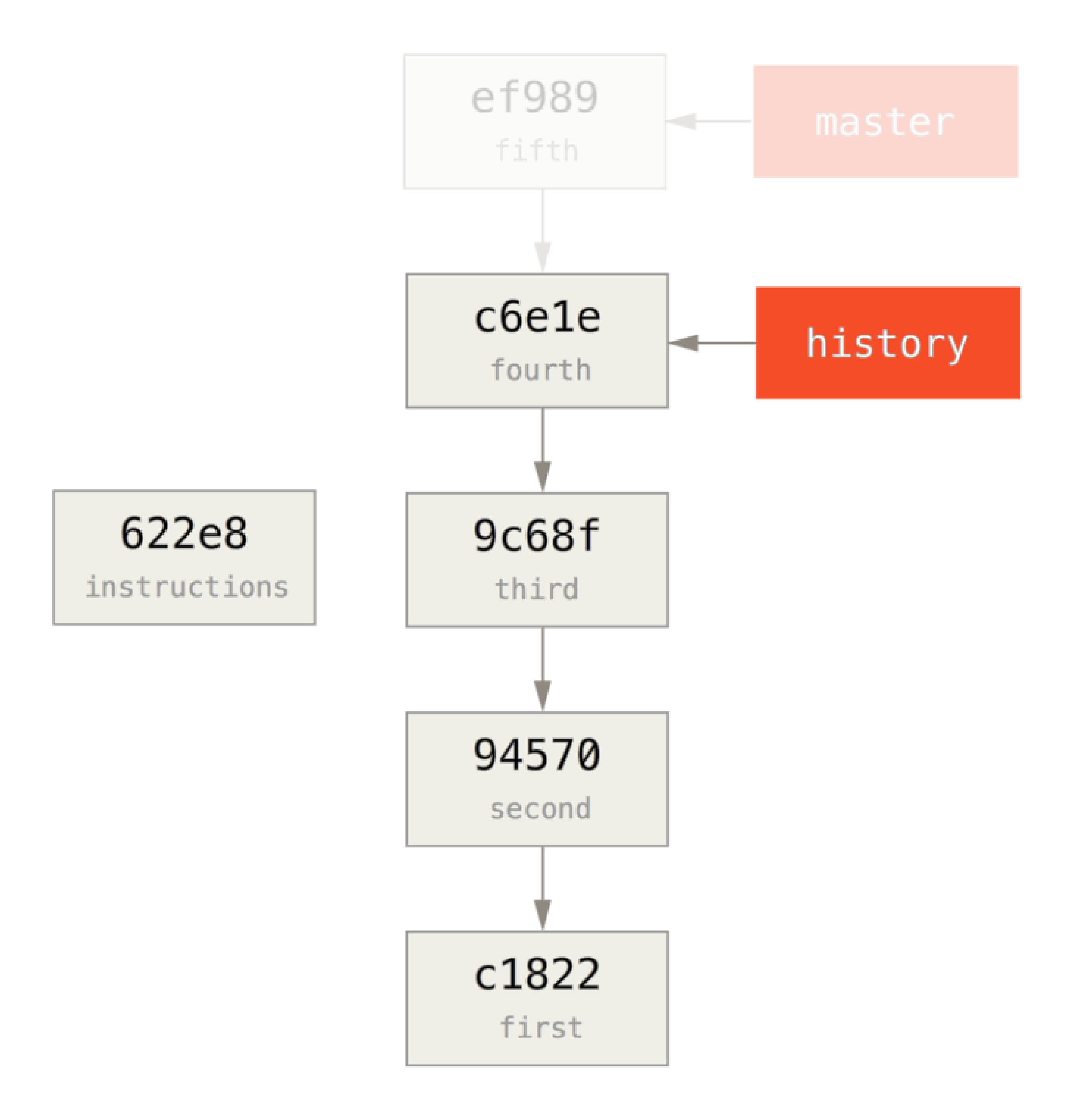
En este caso, es útil crear un commit de base que tenga instrucciones sobre cómo expandir el historial, por lo que otros desarrolladores saben qué hacer si acceden al primer commit en el historial truncado y necesitan más. Por lo tanto, lo que vamos a hacer es crear un objeto de confirmación inicial como nuestro punto base con instrucciones, luego rebase los compromisos restantes (cuatro y cinco) encima de él.

Para ello, debemos elegir un punto para dividir en, que para nosotros es el tercer commit, que es `9c68fdc` en SHA-speak. Por lo tanto, nuestra comisión base se basará en ese árbol. Podemos crear nuestro commit base con el comando `commit-tree`, que solo toma un árbol y nos dará un nuevo objeto de commit sin padres SHA-1.

```
$ echo 'get history from blah blah blah' | git commit-tree 9c68fdc^{tree}
622e88e9cbfbacfb75b5279245b9fb38dfea10cf
```

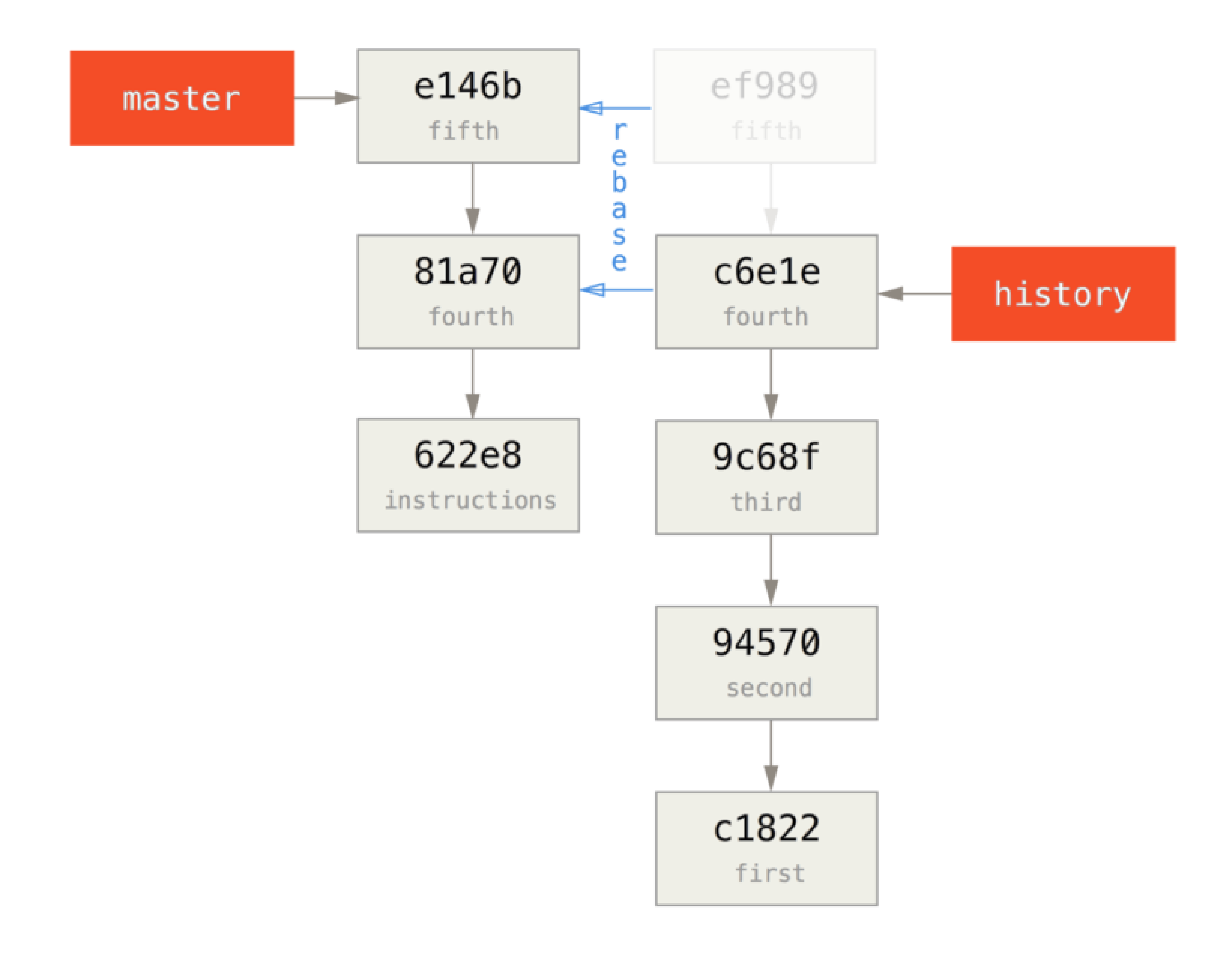
NOTE

El comando `commit-tree` es uno de un conjunto de comandos que comúnmente se denominan comandos *plumbing*. Estos son comandos que no suelen ser utilizados directamente, sino que son utilizados por ** otros comandos Git para hacer trabajos más pequeños. En ocasiones, cuando estamos haciendo cosas más extrañas como estas, nos permiten hacer cosas de nivel muy bajo, pero no son para uso diario. Puede leer más acerca de los comandos de plomería en Fontanería y porcelana



OK, así que ahora que tenemos un commit de base, podemos rebase el resto de nuestra historia encima de eso con 'rebase de git --onto`. El argumento `--onto` será el SHA-1 que acabamos de regresar de` commit-tree` y el punto de rebase será el tercer commit (el padre del primer commit que queremos mantener, `9c68fdc`):

```
$ git rebase --onto 622e88 9c68fdc
First, rewinding head to replay your work on top of it...
Applying: fourth commit
Applying: fifth commit
```



Así que ahora hemos re-escrito nuestra historia reciente en la parte superior de un tiro de base de comisión que ahora tiene instrucciones sobre cómo reconstruir toda la historia si queríamos. Podemos empujar esa nueva historia a un nuevo proyecto y ahora, cuando las personas clonen ese repositorio, solo verán los dos compromisos más recientes y luego un commit de base con instrucciones.

Cambiemos de roles a alguien que clonara el proyecto por primera vez y que quiere toda la historia. Para obtener los datos del historial después de clonar este repositorio truncado, habría que añadir un segundo mando a distancia para el repositorio histórico y buscar:

```
$ git clone https://github.com/schacon/project
$ cd project

$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
622e88e get history from blah blah blah

$ git remote add project-history https://github.com/schacon/project-history
$ git fetch project-history
From https://github.com/schacon/project-history
 * [new branch]      master     -> project-history/master
```

Ahora el colaborador tendría sus compromisos recientes en la rama `master` y los compromisos históricos en la rama `project-history/master`.

```
$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
622e88e get history from blah blah blah

$ git log --oneline project-history/master
c6e1e95 fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

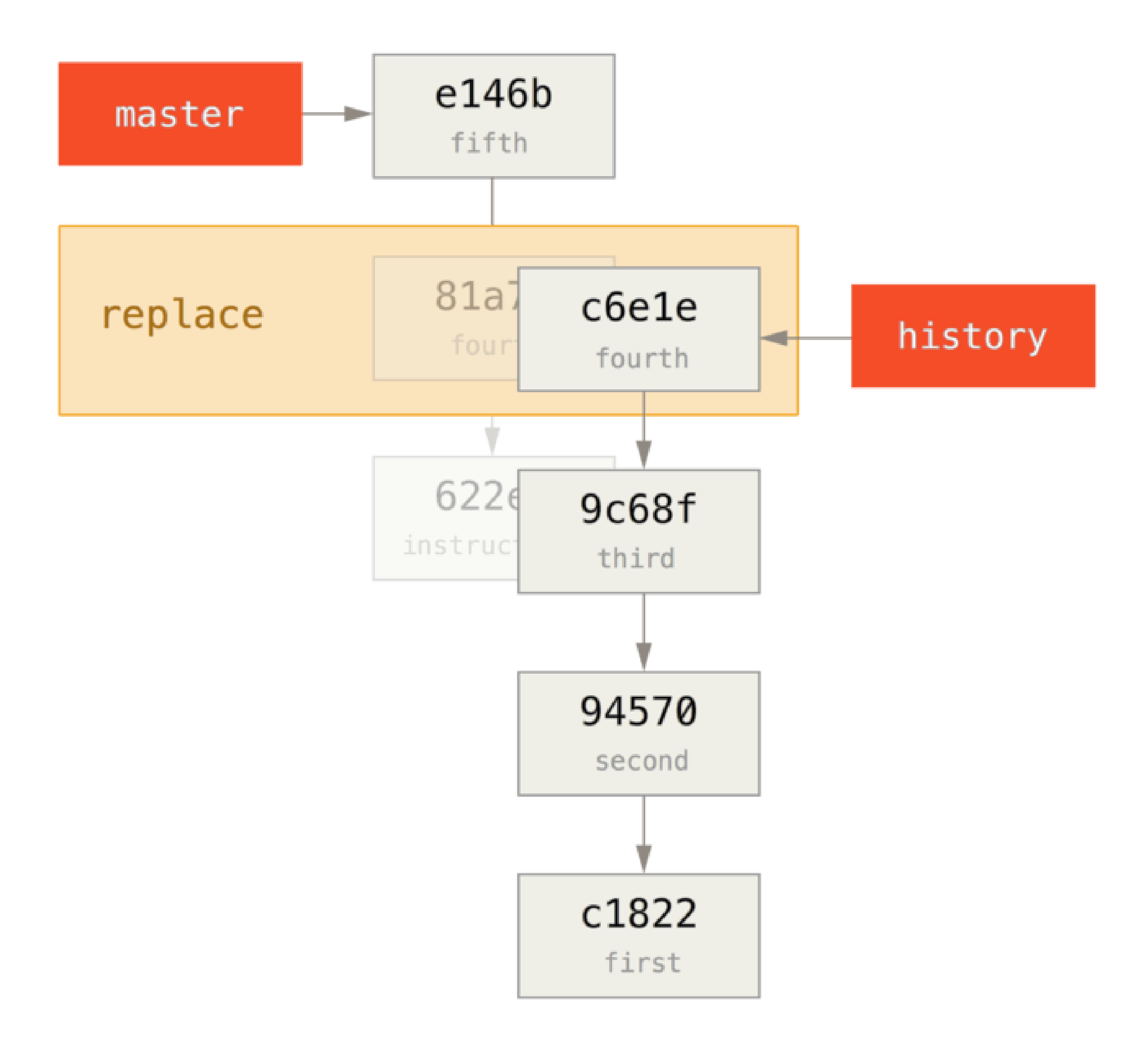
Para combinarlos, simplemente puede llamar a `git replace` con el commit que desea reemplazar y luego el commit con el que desea reemplazarlo. Así que queremos reemplazar el "cuarto" commit en la rama maestra con el "cuarto" commit en la rama `project-history/master`:

```
$ git replace 81a708d c6e1e95
```

Ahora bien, si nos fijamos en la historia de la rama `master`, parece que se ve así:

```
$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

Genial, ¿verdad? Sin tener que cambiar todos los SHA-1s upstream, pudimos reemplazar un commit en nuestra historia con un commit totalmente diferente y todas las herramientas normales (`bisect`,` blame`, etc.) funcionarán como esperamos .



Curiosamente, todavía muestra `81a708d` como el SHA-1, a pesar de que en realidad está utilizando los datos de confirmación` c6e1e95` con los que lo reemplazamos. Incluso si ejecuta un comando como `cat-file`, le mostrará los datos reemplazados:

```
$ git cat-file -p 81a708d
tree 7bc544cf438903b65ca9104a1e30345eee6c083d
parent 9c68fdceee073230f19ebb8b5e7fc71b479c0252
author Scott Chacon <schacon@gmail.com> 1268712581 -0700
committer Scott Chacon <schacon@gmail.com> 1268712581 -0700

fourth commit
```

Recuerde que el padre real de `81a708d` fue nuestro placeholder commit (`622e88e`), no `9c68fdce`, como se indica aquí.

Otra cosa interesante es que estos datos se guardan en nuestras referencias:

```
$ git for-each-ref
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/heads/master
c6e1e95051d41771a649f3145423f8809d1a74d4 commit refs/remotes/history/master
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/remotes/origin/HEAD
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/remotes/origin/master
c6e1e95051d41771a649f3145423f8809d1a74d4 commit
refs/replace/81a708dd0e167a3f691541c7a6463343bc457040
```

Esto significa que es fácil compartir nuestro reemplazo con otros, porque podemos empujar esto a nuestro servidor y otras personas pueden descargarlo fácilmente. Esto no es tan útil en el escenario de injerto de historia que hemos pasado aquí (ya que todo el mundo estaría descargando ambas historias de todos modos, ¿por qué separarlas?), Pero puede ser útil en otras circunstancias.

Almacenamiento de credenciales

Si usa protocolo SSH para conectar a los remotos, es posible tener una llave sin clave, lo que permite tranferir la data sin tener que escribir el nombre de usuario y la clave cada vez. Sin embargo, esto no es posible por el protocolo HTTP - cada conexión necesita usuario y contraseña. Esto se vuelve incluso más complicado para sistemas con autenticación de dos pasos, donde el token que se usa para la clave es generado al azar y no puede ser utilizado.

Afortunadamente, Git tiene un sistema de credenciales que lo ayuda con esto. Git tiene las siguientes funciones disponibles:

- El default es no guardar cache para nada. Cada conexión solicitará el usuario y contraseña.
- El modo “cache” mantiene las credenciales en memoria por un cierto periodo de tiempo. Ninguna de las claves es guardada en disco, y son borradas del cache tras 15 minutos.
- El modo “store” guarda las credenciales en un archivo de texto plano en disco, y nunca expiran. Esto quiere decir que hasta que se cambie la contraseña en el host Git, no se necesitarán escribir credenciales de nuevo. La desventaja de este método es que sus claves son guardadas en texto plano en un archivo dentro de su máquina.
- Si está usando Mac, Git viene con el modo “osxkeychain”, el cual guarda en cache las credenciales en el llavero que está conectado a su cuenta de sistema. Este método guarda las claves en disco, y nunca expiran, pero están encriptadas con el mismo sistema que guarda los certificados HTTPS y los auto-completar de Safari.
- Si está en Windows, puede installer un ayudante llamado “winstore.” Este es similar al ayudante de “osxkeychain” descrito arriba, pero usa Windows Credential Store para controlar la información sensible. Se puede encontrar en https://gitcredentialstore.codeplex.com.

Se puede elegir cualquiera de estos métodos mediante el valor de configuración de Git:

```
$ git config --global credential.helper cache
```

Algunos de estos ayudantes tienen opciones. El modo “store” puede tomar un argumento `--file <ruta>', el cual personaliza la ubicación final del archivo en texto plano (el default es`

`` `~/.git-credentials``).

El modo “cache” acepta la opción `--timeout <segundos>`, la cual cambia la cantidad de tiempo que el demonio se mantiene en ejecución (el default es “900”, o 15 minutos). Aquí hay un ejemplo de cómo configurar el modo “store” con un nombre de archivo personalizado:

```
$ git config --global credential.helper store --file ~/.my-credentials
```

Git incluso permite configurar varios modos. Cuando se buscan por credenciales para un host en particular, Git las mostrará en orden, y se detendrá después que la primer respuesta sea entregada. Cuando se guardan credenciales, Git mandará el usuario y contraseña a **todos** los modos en la lista, y se podrá elegir qué hacer con ellos. Aquí se muestra cómo se vería un archivo `.gitconfig` si tuviera un archivo de credenciales en una memoria, pero quisiera usar lo almacenado en cache cuando la memoria no esté conectada:

```
[credential]
    helper = store --file /mnt/thumbdrive/.git-credentials
    helper = cache --timeout 30000
```

Bajo el sombrero

¿Cómo funciona todo esto? El comando raíz de Git para el asistente de credenciales es `git credential`, el cual toma un comando como argumento, y luego más inputs por medio de stdin.

Esto podría ser más fácil de entender con un ejemplo. Supongamos que un modo de credenciales ha sido configurado, y que el asistente a guardado credenciales para `mygithost`. Aquí hay una sesión que usa el comando “fill”, el cual es invocado cuando Git está intentando encontrar credenciales para el host:

```
$ git credential fill ①
protocol=https ②
host=mygithost
③
protocol=https ④
host=mygithost
username=bob
password=s3cre7
$ git credential fill ⑤
protocol=https
host=unknownhost

Username for 'https://unknownhost': bob
Password for 'https://bob@unknownhost':
protocol=https
host=unknownhost
username=bob
password=s3cre7
```

① Esta es el comando que inicia la interacción.

② Git-credential entonces espera por un input en stdin. Nosotros lo proveemos con lo que conocemos: el protocolo y el nombre de host.

③ Una línea en blanco indica que el input está completo, y el sistema de credencial debería responder con lo que conoce.

④ Git-credential entonces entra en acción, y escribe en stdout los bits de información que encontró.

⑤ Si no se encuentran credenciales, Git pregunta al usuario por el usuario y la contraseña, y lo entrega de vuelta a stdout (aquí ya están conectados a la misma consola).

El sistema de credenciales en realidad está invocando un programa que está separado de Git; el que figura en el valor de configuración `credential.helper`. Hay varias formas que puede tomar:

Configuration Value	**Behavior**
`foo`	Runs `git-credential-foo`
`foo -a --opt=bcd`	Runs `git-credential-foo -a --opt=bcd`
`/absolute/path/foo -xyz`	Runs `/absolute/path/foo -xyz`
`!f() { echo "password=s3cre7"; }; f`	Code after `!` evaluated in shell

Así, los modos descritos arriba en realidad se llaman `git-credential-cache`, `git-credential-store`, y en adelante, y los podemos configurar para que tomen argumentos de línea de comando. La forma general para conseguirlo es “git-credential-foo [args] <acción>.” El protocolo stdin/stdout es el mismo que git-credential, pero usan un conjunto ligeramente distinto de acciones:

- `get` es una petición para un par usuario/contraseña.
- `store` es una petición para guardar un grupo de credenciales en la memoria del modo.

- `erase` purga las credenciales para las propiedades entregadas de la memoria del modo.

Para las acciones `store` y `erase`, no es necesaria una respuesta (Git la ignora de todos modos). Para la acción `get`, sin embargo, Git está muy interesado en lo que el modo tiene que decir. Si el modo no sabe nada útil, puede simplemente salir sin mostrar inforamción, pero si sabe algo, debería aumentar la información provista con la información que ha almacenado. El output es tratado como una serie de declaraciones de assignación; nada provista remplazará lo que Git ya conoce.

Aquí hay un ejemplo de lo explicado, pero saltando git-credential y yendo directo a git-credential-store:

```
$ git credential-store --file ~/git.store store ①
protocol=https
host=mygithost
username=bob
password=s3cre7
$ git credential-store --file ~/git.store get ②
protocol=https
host=mygithost

username=bob ③
password=s3cre7
```

① Aquí decimos con `git-credential-store` que guarde las credenciales: username "bob" y la clave "s3cre7" serán usadas cuando se accese a `https://mygithost`.

② Ahora vamos a recibir las credenciales. Ahora proveemos las partes de la conexión que ya conocemos (`https://mygithost`), y una línea en blanco.

③ `git-credential-store` responde con el usuario y la contraseña que guardamos al comienzo.

Aquí se muestra cómo se vería `~/git.store`:

```
https://bob:s3cre7@mygithost
```

Es solamente una serie de líneas, cada una conteniendo una URL con credenciales. Los modos `osxkeychain` y `winsoter` usan el formato nativo de sus almacenamientos, mientras `cache` usa su propio formato en memoria (el cual no puede ser leído por ningún proceso).

Un cache de credenciales personalizado

Dado que `git-credential-store` y amigos son programas separados de Git, no es difícil de notar que *cualquier* programa puede ser un asistente de credenciales de Git. Los modos provistos por Git cubren muchos de los casos de uso, pero no todos. Por ejemplo, supongamos que tu equipo tiene credenciales que son compartidas con el equipo entero, tal vez para despliegue. Estas están guardadas en un directorio compartido, pero no deseas copiarlas a tu almacén de credenciales, porque cambian de manera seguida. Ninguno de los modos existentes cubre este caso; veamos lo que tomaría para escribir tu propio modo. Existen muchas funcionalidades clave que necesita tener este programa:

1. La única acción que necesitamos vigilar es `get`; `store` y `erase` son operaciones de escritura, así que solo saldremos limpiamente cuando sean recibidas.
2. El formato de archivo de la credencial compartida es el mismo que se usa por `git-credential-store`.
3. La ubicación de ese archivo es relativamente estándar, pero deberías permitir al usuario entregar una ruta alterna por si acaso.

Una vez más, vamos a escribir esta extensión en Ruby, pero cualquier lenguaje funcionará siempre y cuando Git pueda ejecutar el producto final. Aquí está el código de nuestro nuevo asistente de credenciales:

```
#!/usr/bin/env ruby

require 'optparse'

path = File.expand_path '~/.git-credentials' ①
OptionParser.new do |opts|
    opts.banner = 'USAGE: git-credential-read-only [options] <action>'
    opts.on('-f', '--file PATH', 'Specify path for backing store') do |argpath|
        path = File.expand_path argpath
    end
end.parse!

exit(0) unless ARGV[0].downcase == 'get' ②
exit(0) unless File.exists? path

known = {} ③
while line = STDIN.gets
    break if line.strip == ''
    k,v = line.strip.split '=', 2
    known[k] = v
end

File.readlines(path).each do |fileline| ④
    prot,user,pass,host = fileline.scan(/^(.*?):\/\/(.*?):(.*?)@(.*)$/).first
    if prot == known['protocol'] and host == known['host'] then
        puts "protocol=#{prot}"
        puts "host=#{host}"
        puts "username=#{user}"
        puts "password=#{pass}"
        exit(0)
    end
end
```

① Aquí analizamos las opciones de la línea de comando, permitiendo al usuario especificar un archivo. El default es `~/.git-credentials`.

② Este programa solo responde si la acción es `get` y el archivo de almacenamiento existe.

③ Este bucle lee de stdin hasta que se encuentre la primer línea en blanco. Los input son

guardadoes en el hash `known` para una posterior referencia.

④ Este bucle lee el contenido del archivo de almacenamiento, buscando por concordancias. Si el protocolo y el host de `known` concuerdan con la línea, el programa imprime el resultado a stdout y sale.

Guardaremos nuestro modo como `git-credential-read-only`, ponlo en algún lugar en nuestro `PATH` y lo marcamos como ejecutable. Aquí se muestra cómo se vería una sesión interactiva:

```
$ git credential-read-only --file=/mnt/shared/creds get
protocol=https
host=mygithost

protocol=https
host=mygithost
username=bob
password=s3cre7
```

Dado que su nombre comienza con “git-”, podemos usar la sintaxis simple para el valor de configuración:

```
$ git config --global credential.helper read-only --file /mnt/shared/creds
```

Como se puede apreciar, extender este sistema es bastante sencillo, y puede resolver algunos problemas comunes para usted y su equipo.

Resumen

Has visto un número de herramientas avanzadas que te permiten manipular tus commits y el área de staging de una manera más precisa. Cuando notes errores, podrás estar en la capacidad de descubrir fácilmente qué commit lo introdujo, cuándo y por quién. Si quieres usar subproyectos en tu proyecto, has aprendido como acomodar esas necesidades. En este momento, tu debes ser capaz de realizar la mayoría de cosas que vas a necesitar en Git durante tu día a día desde la línea de comandos y sentirte a gusto haciéndolo.

Personalización de Git

Hasta ahora, hemos visto los aspectos básicos del funcionamiento de Git y la manera de utilizarlo; además de haber presentado una serie de herramientas suministradas con Git para ayudarnos a usarlo de manera sencilla y eficiente. En este capítulo, avanzaremos sobre ciertas operaciones que puedes utilizar para personalizar el funcionamiento de Git ; presentando algunos de sus principales ajustes y el sistema de anclajes (hooks). Con estas operaciones, será fácil conseguir que Git trabaje exactamente como tú, tu empresa o tu grupo necesitéis.

Configuración de Git

Como se ha visto brevemente en Inicio - Sobre el Control de Versiones, podemos acceder a los ajustes de configuración de Git a través del comando *git config*. Una de las primeras acciones que has realizado con Git ha sido el configurar tu nombre y tu dirección de correo-e.

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Ahora vas a aprender un puñado de nuevas e interesantes opciones que puedes utilizar para personalizar el uso de Git.

Primeramente, vamos a repasar brevemente los detalles de configuración de Git que ya has visto. Para determinar su comportamiento no estándar, Git emplea una serie de archivos de configuración. El primero de ellos es el archivo `/etc/gitconfig`, que contiene valores para todos y cada uno de los usuarios en el sistema y para todos sus repositorios. Con la opción `--system` del comando `git config`, puedes leer y escribir de/a este archivo.

El segundo es el archivo `~/.gitconfig` (o `~/.config/git/config`), específico para cada usuario. Con la opción `--global`, `git config` lee y escribe en este archivo.

Y por último, Git también puede considerar valores de configuración presentes en el archivo `.git/config` de cada repositorio que estés utilizando. Estos valores se aplicarán únicamente a dicho repositorio.

Cada nivel sobreescribe los valores del nivel anterior; es decir lo configurado en `.git/config` tiene preferencia con respecto a lo configurado en `/etc/gitconfig`, por ejemplo.

NOTE Los archivos de configuración de Git son de texto plano, por lo que también puedes ajustar manualmente los valores de configuración, editando directamente los archivos correspondientes y escribiendo en ellos con la sintaxis correspondiente; pero suele ser más sencillo hacerlo siempre a través del comando `git config`.

Configuración básica del cliente

Las opciones de configuración reconocidas por Git pueden distribuirse en dos grandes categorías: las del lado cliente y las del lado servidor. La mayoría de las opciones están en el lado cliente, – configurando tus preferencias personales de trabajo –. Aunque hay multitud de ellas, aquí vamos a

ver solamente unas pocas, las más comúnmente utilizadas o las que afectan significativamente a tu forma de trabajar. No vamos a revisar aquellas opciones utilizadas solo en casos muy especiales. Si quieres consultar una lista completa, con todas las opciones contempladas en tu versión de Git, puedes lanzar el comando:

```
$ man git-config
```

Este comando muestra todas las opciones con cierto nivel de detalle. También puedes encontrar esta información de referencia en http://git-scm.com/docs/git-config.html.

`core.editor`

Por defecto, Git utiliza cualquier editor que hayas configurado como editor de texto por defecto de tu sistema (`$VISUAL` o `$EDITOR`). O, si no lo has configurado, utilizará `vi` como editor para crear y editar las etiquetas y mensajes de tus confirmaciones de cambio (commit). Para cambiar ese comportamiento, puedes utilizar el ajuste `core.editor`:

```
$ git config --global core.editor emacs
```

A partir de ese comando, por ejemplo, git lanzará Emacs cada vez que vaya a editar mensajes; indistintamente del editor configurado en la línea de comandos (shell) del sistema.

`commit.template`

Si preparas este ajuste para apuntar a un archivo concreto de tu sistema, Git lo utilizará como mensaje por defecto cuando hagas confirmaciones de cambio. Por ejemplo, imagina que creas una plantilla en `~/.gitmessage.txt` con un contenido tal como:

```
subject line

what happened

[ticket: X]
```

Para indicar a Git que lo utilice como mensaje por defecto y que aparezca en tu editor cuando lances el comando `git commit`, tan solo has de ajustar `commit.template`:

```
$ git config --global commit.template ~/.gitmessage.txt
$ git commit
```

A partir de entonces, cada vez que confirmes cambios (commit), tu editor se abrirá con algo como esto:

```
subject line

what happened

[ticket: X]
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# modified:   lib/test.rb
#
~
~
".git/COMMIT_EDITMSG" 14L, 297C
```

Si tienes una política concreta con respecto a los mensajes de confirmación de cambios, puedes aumentar las posibilidades de que sea respetada si creas una plantilla acorde a dicha política y la pones como plantilla por defecto de Git.

`core.pager`

El parámetro core.pager selecciona el paginador utilizado por Git cuando muestra resultados de comandos tales como `log` o `diff`.. Puedes ajustarlo para que utilice `more` o tu paginador favorito, (por defecto, se utiliza `less`); o puedes anular la paginación si le asignas una cadena vacía.

```
$ git config --global core.pager ''
```

Si lanzas esto, Git mostrará siempre el resultado completo de todos los comandos, independientemente de lo largo que sea éste.

`user.signingkey`

Si tienes costumbre de firmar tus etiquetas (tal y como se ha visto en Firmando tu trabajo), configurar tu clave de firma GPG puede facilitarte la labor. Puedes configurar tu clave ID de esta forma:

```
$ git config --global user.signingkey <gpg-key-id>
```

Ahora podrás firmar etiquetas sin necesidad de indicar tu clave cada vez en el comando `git tag`.

```
$ git tag -s <tag-name>
```

`core.excludesfile`

Se pueden indicar expresiones en el archivo *.gitignore* de tu proyecto para indicar a Git lo que debe considerar o no como archivos sin seguimiento, o lo que interará o no seleccionar cuando lances el comando *git add*, tal y como se indicó en Ignorar Archivos.

Pero a veces, necesitas ignorar ciertos archivos en todos los repositorios con los que trabajas. Por ejemplo, si trabajas en una computadora con Mac OS X, estarás al tanto de la existencia de los archivo `.DS_Store`. O si usas Emacs o Vim, también conocerás los archivos terminados en `~`.

Este ajuste puedes grabarlo en un archivo global, llamado `~/.gitignore_global`, con estos contenidos:

```
*~
.DS_Store
```

…y si ahora lanzas `git config --global core.excludesfile ~/.gitignore_global`, Git ya nunca más tendrá en cuenta esos archivos en tus repositorios.

`help.autocorrect`

Si te equivocas al teclear un comando de Git, te mostrará algo como:

```
$ git chekcout master
git: 'chekcout' is not a git command. See 'git --help'.

Did you mean this?
    checkout
```

Git intenta imaginar qué es lo que querías escribir, pero aun así no lo intenta ejecutar. Si pones la opción `help.autocorrect` a 1, Git sí lanzará el comando corrigiendo tu error:

```
$ git chekcout master
WARNING: You called a Git command named 'chekcout', which does not exist.
Continuing under the assumption that you meant 'checkout'
in 0.1 seconds automatically...
```

Observa lo de “0.1 seconds”. Es un entero que representa décimas de segundo. Si le das un valor 50, Git retrasará la ejecución final del comando 5 segundos con el fin de que puedas abortar la operación auto-corregida con la opción `help.autocorrect`.

Colores en Git

Git puede marcar con colores los resultados que muestra en tu terminal, ayudándote así a leerlos más fácilmente. Hay unos cuantos parámetros que te pueden ayudar a configurar tus colores favoritos.

`color.ui`

Si se lo pides, Git coloreará automáticamente la mayor parte de los resultados que muestre. Puedes ajustar con precisión cada una de las partes a colorear; pero si deseas activar de un golpe todos los colores por defecto, no tienes más que poner a "true" el parámetro *color.ui*.

Para desactivar totalmente los colores, puedes hacer esto:

```
$ git config --global color.ui false
```

El valor predeterminado es `auto`, que colorea la salida cuando va a un terminal, pero no lo hace cuando se envía la salida a un archivo o a una tubería.

También puedes ponerlo a `always` para hacer que se coloree siempre. Es muy raro que quieras hacer esto, ya que cuando se quiere puntualmente colorear la salida redirigida se puede pasar un flag `--color` al comando Git.

`color.*`

Cuando quieras ajustar específicamente, comando a comando, donde colorear y cómo colorear, puedes emplear los ajustes particulares de color. Cada uno de ellos puede fijarse a `true` (verdadero), `false` (falso) o `always` (siempre):

```
color.branch
color.diff
color.interactive
color.status
```

Además, cada uno de ellos tiene parámetros adicionales para asignar colores a partes específicas, por si quieres precisar aún más. Por ejemplo, para mostrar la meta-información del comando `diff` con letra azul sobre fondo negro y con caracteres en negrita, puedes indicar:

```
$ git config --global color.diff.meta "blue black bold"
```

Puedes ajustar un color a cualquiera de los siguientes valores: `normal`, `black` (negro), `red` (rojo), `green` (verde), `yellow` (amarillo), `blue` (azul), `magenta`, `cyan` (cian), o `white` (blanco).

También puedes aplicar atributos tales como `bold` (negrita), `dim` (tenue), `ul` (subrayado), `blink` (parpadeante) y `reverse`(video inverso).

Herramientas externas para fusión y diferencias

Aunque Git lleva una implementación interna de diff, la que se utiliza habitualmente, se puede sustituir por una herramienta externa. Puedes incluso configurar una herramienta gráfica para la resolución de conflictos, en lugar de resolverlos manualmente. Lo voy a demostrar configurando Perforce Visual Merge (P4Merge) como herramienta para realizar las comparaciones y resolver conflictos; ya que es una buena herramienta gráfica y es libre.

Si lo quieres probar, P4Merge funciona en todas las principales plataformas. Los nombres de carpetas que utilizaré en los ejemplos funcionan en sistemas Mac y Linux; para Windows, tendrás que sustituir */usr/local/bin* por el correspondiente camino al ejecutable en tu sistema.

P4Merge se puede descargar desde http://www.perforce.com/downloads/Perforce/.

Para empezar, tienes que preparar los correspondientes scripts para lanzar tus comandos. En estos ejemplos, voy a utilizar caminos y nombres Mac para los ejecutables; en otros sistemas, tendrás que sustituirlos por los correspondientes donde tengas instalado *p4merge*. El primer script a preparar es uno al que denominaremos *extMerge*, para llamar al ejecutable con los correspondientes argumentos:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/p4merge.app/Contents/MacOS/p4merge $*
```

El script para el comparador, ha de asegurarse de recibir siete argumentos y de pasar dos de ellos al script de fusión (merge). Por defecto, Git pasa los siguientes argumentos al programa diff (comparador):

```
path old-file old-hex old-mode new-file new-hex new-mode
```

Ya que solo necesitarás *old-file* y *new-file*, puedes utilizar el siguiente script para extraerlos:

```
$ cat /usr/local/bin/extDiff
#!/bin/sh
[ $# -eq 7 ] && /usr/local/bin/extMerge "$2" "$5"
```

Además has de asegurarte de que estas herramientas son ejecutables:

```
$ sudo chmod +x /usr/local/bin/extMerge
$ sudo chmod +x /usr/local/bin/extDiff
```

Una vez preparado todo esto, puedes ajustar el archivo de configuración para utilizar tus herramientas personalizadas de comparación y resolución de conflictos. Tenemos varios parámetros a ajustar: `merge.tool` para indicar a Git la estrategia que ha de usar, `mergetool.*.cmd` para especificar como lanzar el comando, `mergetool.trustExitCode` para decir a Git si el código de salida del programa indica una fusión con éxito o no, y `diff.external` para decir a Git qué comando lanzar para realizar comparaciones. Es decir, has de ejecutar cuatro comandos de configuración:

```
$ git config --global merge.tool extMerge
$ git config --global mergetool.extMerge.cmd \
  'extMerge \"$BASE\" \"$LOCAL\" \"$REMOTE\" \"$MERGED\"'
$ git config --global mergetool.extMerge.trustExitCode false
$ git config --global diff.external extDiff
```

o puedes editar tu archivo *~/.gitconfig* para añadirle las siguientes líneas:

```
[merge]
  tool = extMerge
[mergetool "extMerge"]
  cmd = extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"
  trustExitCode = false
[diff]
  external = extDiff
```

Tras ajustar todo esto, si lanzas comandos tales como:

```
$ git diff 32d1776b1^ 32d1776b1
```

En lugar de mostrar las diferencias por línea de comandos, Git lanzará P4Merge, que tiene una pinta como:

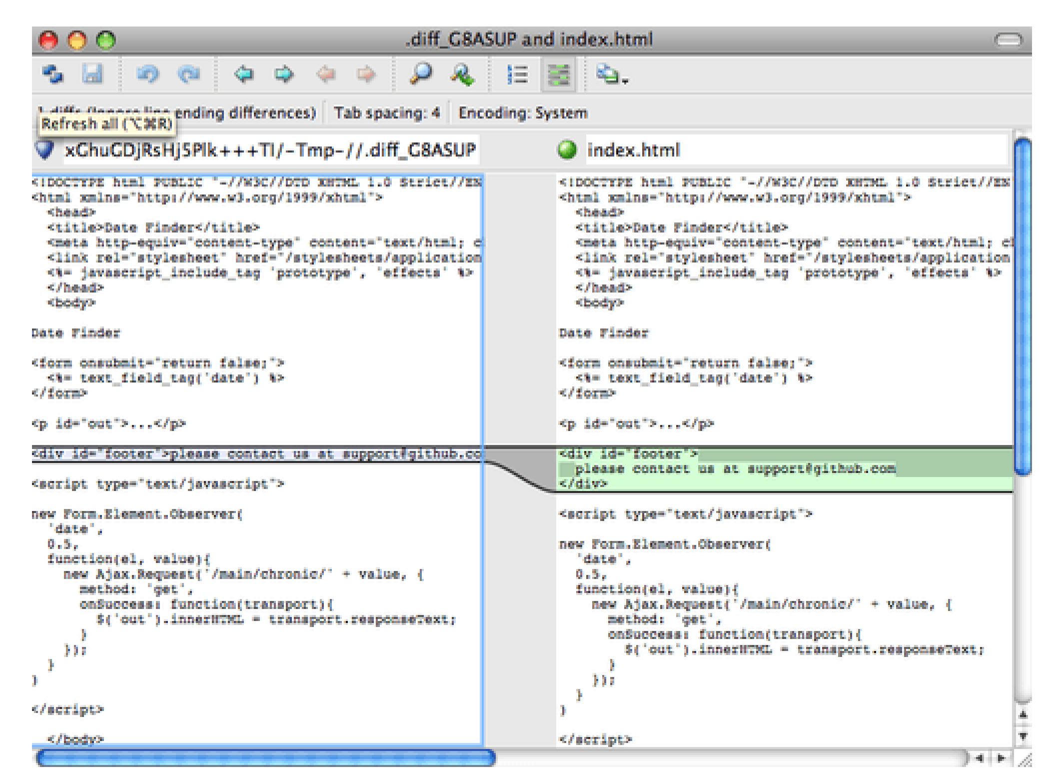


Figure 143. P4Merge.

Si intentas fusionar (merge) dos ramas y tienes los consabidos conflictos de integración, puedes lanzar el comando `git mergetool`; lanzará P4Merge para ayudarte a resolver los conflictos por medio de su interfaz gráfica.

Lo bonito de estos ajustes con scripts, es que puedes cambiar fácilmente tus herramientas de comparación (diff) y de fusión (merge). Por ejemplo, para cambiar tus scripts `extDiff` y `extMerge` para utilizar KDiff3, tan solo has de editar el archivo `extMerge`:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/kdiff3.app/Contents/MacOS/kdiff3 $*
```

A partir de ahora, Git utilizará la herramienta KDiff3 para mostrar y resolver conflictos de integración.

Git viene preparado para utilizar bastantes otras herramientas de resolución de conflictos, sin necesidad de andar ajustando la configuración. Para listar las herramientas soportadas solo has de lanzar el comando:

```
$ git mergetool --tool-help
'git mergetool --tool=<tool>' may be set to one of the following:
        emerge
        gvimdiff
        gvimdiff2
        opendiff
        p4merge
        vimdiff
        vimdiff2

The following tools are valid, but not currently available:
        araxis
        bc3
        codecompare
        deltawalker
        diffmerge
        diffuse
        ecmerge
        kdiff3
        meld
        tkdiff
        tortoisemerge
        xxdiff

Some of the tools listed above only work in a windowed
environment. If run in a terminal-only session, they will fail.
```

Si no te interesa utilizar KDiff3 para comparaciones, sino que tan solo te interesa utilizarlo para resolver conflictos de integración; teniendo kdiff3 en el path de ejecución, solo has de lanzar el comando:

```
$ git config --global merge.tool kdiff3
```

Si utilizas este comando en lugar de preparar los archivos `extMerge` y `extDiff` antes comentados, Git utilizará KDiff3 para resolución de conflictos de integración y la herramienta estándar diff para las comparaciones.

Formateo y espacios en blanco

El formato y los espacios en blanco son la fuente de los problemas más sutiles y frustrantes que muchos desarrolladores se pueden encontrar en entornos colaborativos, especialmente si son multi-plataforma. Es muy fácil que algunos parches u otro trabajo recibido introduzcan sutiles cambios de espaciado, porque los editores suelen hacerlo inadvertidamente o, trabajando en entornos multi-plataforma, porque programadores Windows suelen añadir retornos de carro al final de las lineas que tocan. Git dispone de algunas opciones de configuración para ayudarnos con estos problemas.

`core.autocrlf`

Si estás programando en Windows o utilizando algún otro sistema, pero colaborando con gente que programa en Windows. Es muy posible que alguna vez te topes con problemas de finales de línea. Esto se debe a que Windows utiliza retorno-de-carro y salto-de-linea para marcar los finales de línea de sus archivos. Mientras que Mac y Linux utilizan solamente el carácter de salto-de-linea. Esta es una sutil, pero molesta, diferencia cuando se trabaja en entornos multi-plataforma.

Git la maneja convirtiendo automáticamente los finales CRLF en LF al hacer confirmaciones de cambios (commit); y, viceversa, al extraer código (checkout) a la carpeta de trabajo. Puedes activar esta funcionalidad con el parámetro `core.autocrlf`. Si estás trabajando en una máquina Windows, ajustalo a `true`, para convertir finales LF en CRLF cuando extraigas código (checkout), puedes hacer esto:

```
$ git config --global core.autocrlf true
```

Si estás trabajando en una máquina Linux o Mac, entonces no te interesa convertir automáticamente los finales de línea al extraer código, sino que te interesa arreglar los posibles CRLF que pudieran aparecer accidentalmente. Puedes indicar a Git que convierta CRLF en LF al confirmar cambios (commit), pero no en el otro sentido; utilizando también el parámetro `core.autocrlf`:

```
$ git config --global core.autocrlf input
```

Este ajuste dejará los finales de línea CRLF en las extracciones de código (checkout), pero dejará los finales LF en sistemas Mac o Linux, y en el repositorio.

Si eres un programador Windows, trabajando en un entorno donde solo haya máquinas Windows, puedes desconectar esta funcionalidad, para almacenar CRLFs en el repositorio. Ajustando el parámero a `false`:

```
$ git config --global core.autocrlf false
```

`core.whitespace`

Git viene preajustado para detectar y resolver algunos de los problemas más típicos relacionados con los espacios en blanco. Puede vigilar acerca de seis tipos de problemas de espaciado: tres los tiene activados por defecto, pero se pueden desactivar; y tres vienen desactivados por defecto, pero se pueden activar.

Los que están activos de forma predeterminada son `blank-at-eol`, que busca espacios al final de la línea; `blank-at-eof`, que busca líneas en blanco al final del archivo y `espace-before-tab`, que busca espacios delante de las tabulaciones al comienzo de una línea.

Los que están inactivos de forma predeterminada son `ident-with-non-tab`, que busca líneas que empiezan con espacios en blanco en lugar de tabulaciones (y se controla con la opción `tabwidth`); `tab-in-indent`, que busca tabulaciones en el trozo indentado de una línea; y `cr-at-eol`, que informa a Git de que los CR al final de las líneas son correctos.

Puedes decir a Git cuales de ellos deseas activar o desactivar, ajustando el parámetro `core.whitespace` con los valores on/off separados por comas. Puedes desactivarlos tanto dejándolos fuera de la cadena de ajustes, como añadiendo el prefijo `-` delante del valor. Por ejemplo, si deseas activar todos menos `cr-at-eol` puedes lanzar:

```
$ git config --global core.whitespace \
    trailing-space,space-before-tab,indent-with-non-tab
```

Git detectará posibles problemas cuando lance un comando *git diff*, e intentará destacarlos en otro color para que puedas corregirlos antes de confirmar cambios (commit). También pueden ser útiles estos ajustes cuando estás incorporando parches con *git apply*. Al incorporar parches, puedes pedirle a Git que te avise específicamente sobre determinados problemas de espaciado:

```
$ git apply --whitespace=warn <patch>
```

O puedes pedirle que intente corregir automáticamente los problemas antes de aplicar el parche:

```
$ git apply --whitespace=fix <patch>
```

Estas opciones se pueden aplicar también al comando `git rebase`. Si has confirmado cambios con problemas de espaciado, pero no los has enviado (push) aún "aguas arriba" (upstream). Puedes realizar una reorganización (rebase) con la opción `--whitespace=fix` para que Git corrija automáticamente los problemas según va reescribiendo los parches.

Configuración del servidor

No hay tantas opciones de configuración en el lado servidor de Git, pero hay unas pocas interesantes que merecen ser tenidas en cuenta.

`receive.fsckObjects`

Por defecto, Git no suele comprobar la consistencia de todos los objetos que recibe durante un envío (push), aunque Git tiene la capacidad para asegurarse de que cada objeto sigue casando con su suma de control SHA-1 y sigue apuntando a objetos válidos. No lo suele hacer en todos y cada uno de los envíos (push). Es una operación costosa, que, dependiendo del tamaño del repositorio, puede llegar a añadir mucho tiempo a cada operación de envío (push). De todas formas, si deseas que Git compruebe la consistencia de todos los objetos en todos los envíos, puedes forzarle a hacerlo ajustando a *true* el parámetro `receive.fsckObjects`:

```
$ git config --system receive.fsckObjects true
```

A partir de ese momento, Git comprobará la integridad del repositorio antes de aceptar ningún envío (push), para asegurarse de que no está introduciendo datos corruptos.

`receive.denyNonFastForwards`

Si reorganizas (rebase) confirmaciones de cambio (commit) que ya habías enviado y tratas de enviarlas (push) de nuevo; o si intentas enviar una confirmación a una rama remota que no contiene la confirmación actualmente apuntada por la rama, normalmente, la operación te será denegada por la rama remota sobre la que pretendías realizarla. Habitualmente, este es el comportamiento más adecuado. Pero, en el caso de las reorganizaciones, cuando estás totalmente seguro de lo que haces, puedes forzar el envío, utilizando la opción `-f` en el comando `git push` a la rama remota.

Para impedir estos envíos forzados de referencias de avance no directo (no fast-forward) a ramas remotas, es para lo que se emplea el parámetro `receive.denyNonFastForwards`:

```
$ git config --system receive.denyNonFastForwards true
```

Otra manera de obtener el mismo resultado, es a través de los enganches (hooks) en el lado servidor, enganches de los que hablaremos en breve. Esta otra vía te permite realizar ajustes más finos, tales como denegar referencias de avance no directo, (non-fast-forwards), únicamente a un grupo de usuarios.

`receive.denyDeletes`

Uno de los cortocircuitos que suelen utilizar los usuarios para saltarse la politica de `denyNonFastForwards`, suele ser el borrar la rama y luego volver a enviarla de vuelta con la nueva referencia. Puedes evitar esto poniendo a `true` el parámetro `receive.denyDeletes`:

```
$ git config --system receive.denyDeletes true
```

Esto impide el borrado de ramas o de etiquetas por medio de un envío a través de la mesa (push across the board), --ningún usuario lo podrá hacer--. Para borrar ramas remotas, tendrás que borrar los archivos de referencia manualmente sobre el propio servidor. Existen también algunas otras maneras más interesantes de hacer esto mismo, pero para usuarios concretos, a través de

permisos (ACLs); tal y como veremos en Un ejemplo de implantación de una determinada política en Git.

Git Attributes

Algunos de los ajustes que hemos visto, pueden ser especificados para un camino (path) concreto, de tal forma que Git los aplicará únicamente para una carpeta o para un grupo de archivos determinado. Estos ajustes específicos relacionados con un camino, se denominan atributos en Git. Y se pueden fijar, bien mediante un archivo `.gitattribute` en uno de los directorios de tu proyecto (normalmente en la raíz del proyecto), o bien mediante el archivo `git/info/attributes` en el caso de no querer guardar el archivo de atributos dentro de tu proyecto.

Por medio de los atributos, puedes hacer cosas tales como indicar diferentes estrategias de fusión para archivos o carpetas concretas de tu proyecto, decirle a Git cómo comparar archivos no textuales, o indicar a Git que filtre ciertos contenidos antes de guardarlos o de extraerlos del repositorio Git. En esta sección, aprenderás acerca de algunos atributos que puedes asignar a ciertos caminos (paths) dentro de tu proyecto Git, viendo algunos ejemplos de cómo utilizar sus funcionalidades de manera práctica.

Archivos binarios

Un buen truco donde utilizar los atributos Git es para indicarle cuáles de los archivos son binarios (en los casos en que Git no podría llegar a determinarlo por sí mismo), dándole a Git instrucciones especiales sobre cómo tratar estos archivos. Por ejemplo, algunos archivos de texto se generan automáticamente y no tiene sentido compararlos; mientras que algunos archivos binarios sí que pueden ser comparados. Vamos a ver cómo indicar a Git cuál es cuál.

Identificación de archivos binarios

Algunos archivos aparentan ser textuales, pero a efectos prácticos merece más la pena tratarlos como binarios. Por ejemplo, los proyectos Xcode en un Mac contienen un archivo terminado en `.pbxproj`. Este archivo es básicamente una base de datos JSON (datos Javascript en formato de texto plano), escrita directamente por el IDE para almacenar aspectos tales como tus ajustes de compilación. Aunque técnicamente es un archivo de texto (ya que su contenido lo forman caracteres UTF-8). Realmente nunca lo tratarás como tal, porque en realidad es una base de datos ligera (y no puedes fusionar sus contenidos si dos personas lo cambian, porque las comparaciones no son de utilidad). Éstos son archivos destinados a ser tratados de forma automatizada. Y es preferible tratarlos como si fueran archivos binarios.

Para indicar a Git que trate todos los archivos `pbxproj` como binarios, puedes añadir esta línea a tu archivo `.gitattributes`:

```
*.pbxproj binary
```

A partir de ahora, Git no intentará convertir ni corregir problemas CRLF en los finales de línea; ni intentará hacer comparaciones ni mostar diferencias de este archivo cuando lances comandos `git show` o `git diff` en tu proyecto.

Comparación entre archivos binarios

Puedes utilizar los atributos Git para comparar archivos binarios. Se consigue diciéndole a Git la forma de convertir los datos binarios en texto, consiguiendo así que puedan ser comparados con la herramienta habitual de comparación textual.

En primer lugar, utilizarás esta técnica para resolver uno de los problemas más engorrosos conocidos por la humanidad: el control de versiones en documentos Word. Todo el mundo conoce el hecho de que Word es el editor más horroroso de cuantos hay; pero, desgraciadamente, todo el mundo lo usa. Si deseas controlar versiones en documentos Word, puedes añadirlos a un repositorio Git e ir realizando confirmaciones de cambio (commit) cada vez. Pero, ¿qué ganas con ello?. Si lanzas un comando `git diff`, lo único que verás será algo tal como:

```
$ git diff
diff --git a/chapter1.docx b/chapter1.docx
index 88839c4..4afcb7c 100644
Binary files a/chapter1.docx and b/chapter1.docx differ
```

No puedes comparar directamente dos versiones, a no ser que extraigas ambas y las compares manualmente, ¿no?. Pero resulta que puedes hacerlo bastante mejor utilizando los atributos Git. Poniendo lo siguiente en tu archivo `.gitattributes`:

```
*.docx diff=word
```

Así decimos a Git que sobre cualquier archivo coincidente con el patrón indicado, (.docx), ha de utilizar el filtro "word" cuando intente hacer una comparación con él. ¿Qué es el filtro "word"? Tienes que configurarlo tú mismo. Por ejemplo, puedes configurar Git para que utilice el programa `docx2txt` para convertir los documentos Word en archivos de texto planos, archivos sobre los que poder realizar comparaciones sin problemas.

En primer lugar, necesitas instalar `docx2txt`, obteniéndolo de: http://docx2txt.sourceforge.net. Sigue las instrucciones del archivo `INSTALL` e instálalo en algún sitio donde se pueda ejecutar desde la shell. A continuación, escribe un script que sirva para convertir el texto al formato que Git necesita, utilizando docx2txt:

```
#!/bin/bash
docx2txt.pl $1 -
```

No olvides poner los permisos de ejecución al script (`chmod a+x`). Finalmente, configura Git para usar el script:

```
$ git config diff.word.textconv docx2txt
```

Ahora Git ya sabrá que si intentas comparar dos archivos, y cualquiera de ellos finaliza en `.docx`, lo hará a través del filtro "word", que se define con el programa `docx2txt`. Esto provoca la creación de

versiones de texto de los archivos Word antes de intentar compararlos.

Por ejemplo, el capítulo 1 de este libro se convirtió a Word y se envió al repositorio Git. Cuando añadimos posteriormente un nuevo párrafo, el `git diff` muestra lo siguiente:

```
$ git diff
diff --git a/chapter1.docx b/chapter1.docx
index 0b013ca..ba25db5 100644
--- a/chapter1.docx
+++ b/chapter1.docx
@@ -2,6 +2,7 @@
 This chapter will be about getting started with Git. We will begin at the beginning
by explaining some background on version control tools, then move on to how to get Git
running on your system and finally how to get it setup to start working with. At the
end of this chapter you should understand why Git is around, why you should use it and
you should be all setup to do so.
 1.1. About Version Control
 What is "version control", and why should you care? Version control is a system that
records changes to a file or set of files over time so that you can recall specific
versions later. For the examples in this book you will use software source code as the
files being version controlled, though in reality you can do this with nearly any type
of file on a computer.
+Testing: 1, 2, 3.
 If you are a graphic or web designer and want to keep every version of an image or
layout (which you would most certainly want to), a Version Control System (VCS) is a
very wise thing to use. It allows you to revert files back to a previous state, revert
the entire project back to a previous state, compare changes over time, see who last
modified something that might be causing a problem, who introduced an issue and when,
and more. Using a VCS also generally means that if you screw things up or lose files,
you can easily recover. In addition, you get all this for very little overhead.
 1.1.1. Local Version Control Systems
 Many people's version-control method of choice is to copy files into another
directory (perhaps a time-stamped directory, if they're clever). This approach is very
common because it is so simple, but it is also incredibly error prone. It is easy to
forget which directory you're in and accidentally write to the wrong file or copy over
files you don't mean to.
```

Git nos muestra que se añadió la línea “Testing: 1, 2, 3.”, lo cual es correcto. No es perfecto (los cambios de formato, por ejemplo, no los mostrará) pero sirve en la mayoría de los casos.

Otro problema donde puede ser útil esta técnica, es en la comparación de imágenes. Un camino puede ser pasar los archivos JPEG a través de un filtro para extraer su información EXIF (los metadatos que se graban dentro de la mayoría de formatos gráficos). Si te descargas e instalas el programa `exiftool`, puedes utilizarlo para convertir tus imágenes a textos (metadatos), de tal forma que diff podrá al menos mostrarte algo útil de cualquier cambio que se produzca:

```
$ echo '*.png diff=exif' >> .gitattributes
$ git config diff.exif.textconv exiftool
```

Si reemplazas cualquier imagen en el proyecto y ejecutas `git diff`, verás algo como:

```
diff --git a/image.png b/image.png
index 88839c4..4afcb7c 100644
--- a/image.png
+++ b/image.png
@@ -1,12 +1,12 @@
 ExifTool Version Number         : 7.74
-File Size                       : 70 kB
-File Modification Date/Time     : 2009:04:21 07:02:45-07:00
+File Size                       : 94 kB
+File Modification Date/Time     : 2009:04:21 07:02:43-07:00
 File Type                       : PNG
 MIME Type                       : image/png
-Image Width                     : 1058
-Image Height                    : 889
+Image Width                     : 1056
+Image Height                    : 827
 Bit Depth                       : 8
 Color Type                      : RGB with Alpha
```

Aquí se ve claramente que ha cambiado el tamaño del archivo y las dimensiones de la imagen.

Expansión de palabras clave

Algunos usuarios de sistemas SVN o CVS, echan de menos el disponer de expansiones de palabras clave al estilo de las que dichos sistemas tienen. El principal problema para hacerlo en Git reside en la imposibilidad de modificar los archivos con información relativa a la confirmación de cambios (commit). Debido a que Git calcula sus sumas de comprobación antes de las confirmaciones. De todas formas, es posible inyectar textos en un archivo cuando lo extraemos del repositorio (checkout) y quitarlos de nuevo antes de devolverlo al repositorio (commit). Los atributos Git admiten dos maneras de realizarlo.

La primera, es inyectando automáticamente la suma de comprobación SHA-1 de un gran objeto binario (blob) en un campo *Id* dentro del archivo. Si colocas este atributo en un archivo o conjunto de archivos, Git lo sustituirá por la suma de comprobación SHA-1 la próxima vez que lo/s extraiga/s. Es importante destacar que no se trata de la suma SHA de la confirmación de cambios (commit), sino del propio objeto binario (blob):

```
$ echo '*.txt ident' >> .gitattributes
$ echo '$Id$' > test.txt
```

La próxima vez que extraigas el archivo, Git le habrá inyectado el SHA-1 del objeto binario (blob):

```
$ rm test.txt
$ git checkout -- test.txt
$ cat test.txt
$Id: 42812b7653c7b88933f8a9d6cad0ca16714b9bb3 $
```

Pero esto tiene un uso bastante limitado. Si has utilizado alguna vez las sustituciones de CVS o de Subversion, sabrás que pueden incluir una marca de fecha (la suma de comprobación SHA no es igual de util, ya que, por ser bastante aleatoria, es imposible deducir si una suma SHA es anterior o posterior a otra).

Aunque resulta que también puedes escribir tus propios filtros para realizar sustituciones en los archivos al guardar o recuperar (commit/checkout). Se trata de los filtros “clean” y “smudge”. En el archivo ‘.gitattibutes’ puedes indicar filtros para carpetas o archivos determinados y luego preparar tus propios scripts para procesarlos justo antes de confirmar cambios en ellos (“clean”, ver Ejecución de filtro “smudge” en el checkout.), o justo antes de recuperarlos (“smudge”, ver Ejecución de filtro “clean” antes de confirmar el cambio.). Estos filtros pueden utilizarse para realizar todo tipo de acciones útiles.

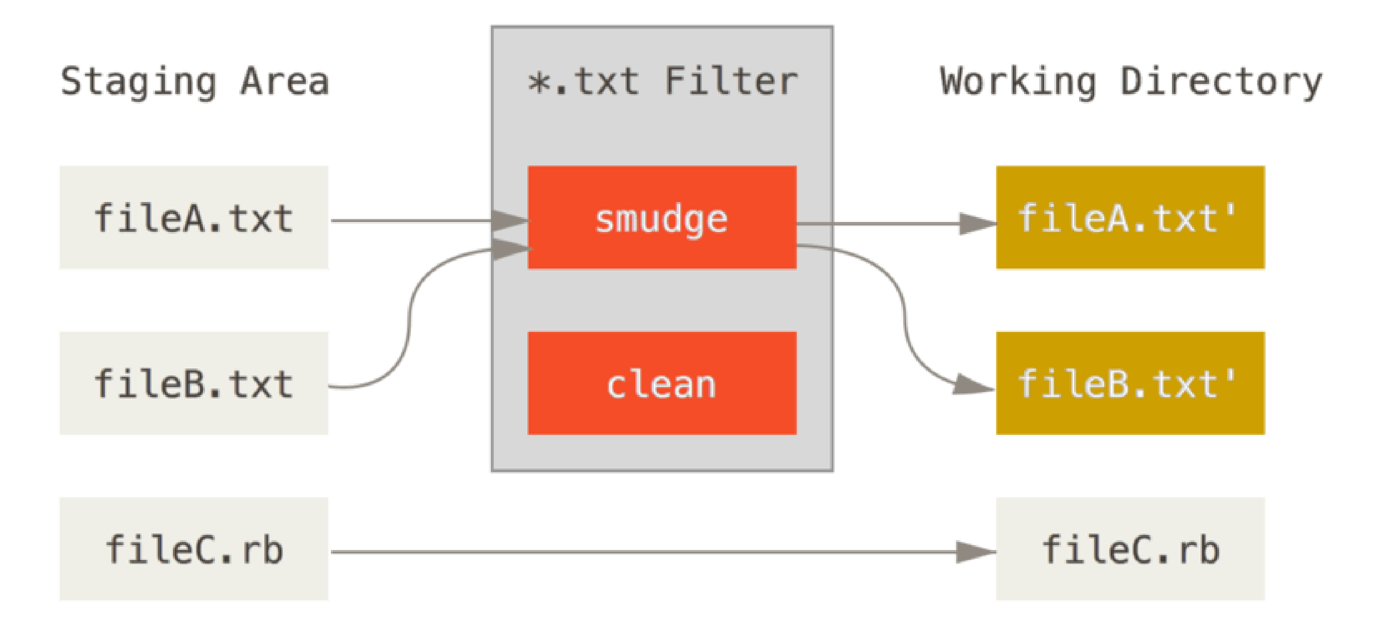


Figure 144. Ejecución de filtro “smudge” en el checkout.

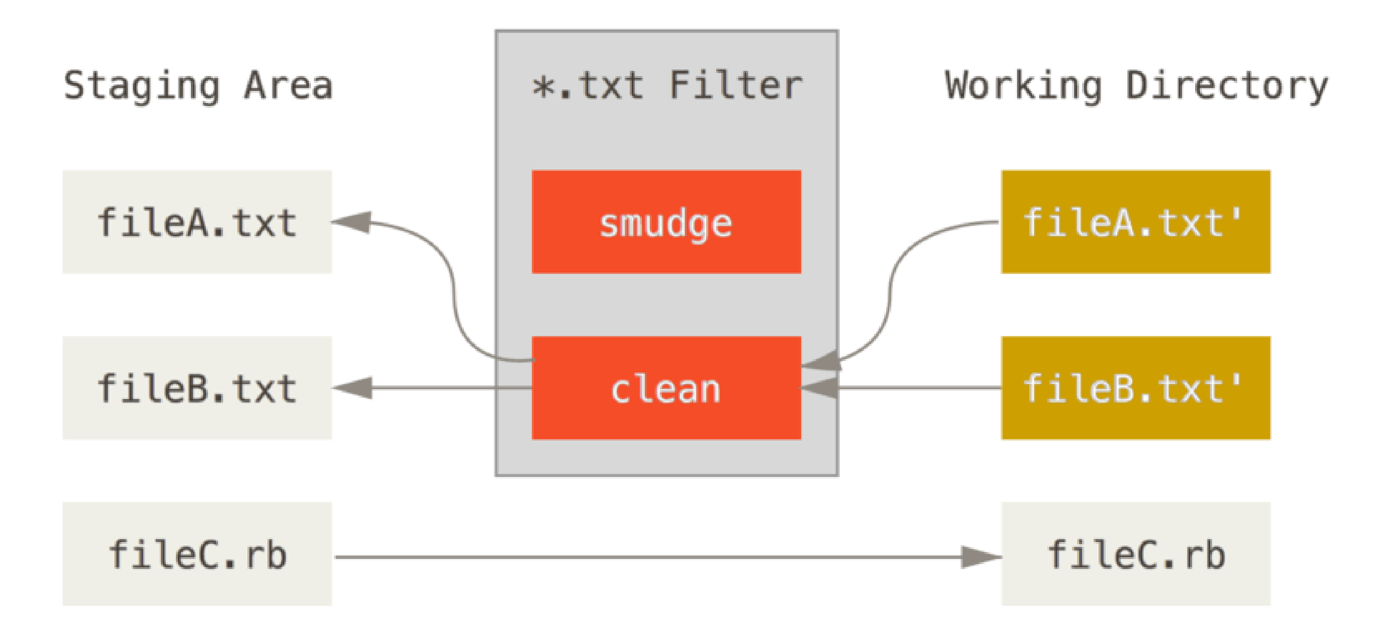


Figure 145. Ejecución de filtro “clean” antes de confirmar el cambio.

El mensaje de confirmación para esta funcionalidad nos da un ejemplo simple: el de pasar todo tu código fuente C por el programa `indent` antes de almacenarlo. Puedes hacerlo poniendo los atributos adecuados en tu archivo `.gitattributes`, para filtrar los archivos *.c a través de “indent”:

```
*.c filter=indent
```

E indicando después que el filtro “indent” actuará al manchar (smudge) y al limpiar (clean):

```
$ git config --global filter.indent.clean indent
$ git config --global filter.indent.smudge cat
```

En este ejemplo, cuando confirmes cambios (commit) en archivos con extensión `*.c`, Git los pasará previamente a través del programa `indent` antes de confirmarlos, y los pasará a través del programa `cat` antes de extraerlos de vuelta al disco. El programa `cat` es básicamente transparente: de él salen los mismos datos que entran. El efecto final de esta combinación es el de filtrar todo el código fuente C a través de `indent` antes de confirmar cambios en él.

Otro ejemplo interesante es el de poder conseguir una expansión de la clave `$Date$` del estilo de RCS. Para hacerlo, necesitas un pequeño script que coja el nombre de un archivo, localice la fecha de la última confirmación de cambios en el proyecto, e inserte dicha información en el archivo. Este podria ser un pequeño script Ruby para hacerlo:

```
#! /usr/bin/env ruby
data = STDIN.read
last_date = `git log --pretty=format:"%ad" -1`
puts data.gsub('$Date$', '$Date: ' + last_date.to_s + '$')
```

Simplemente, utiliza el comando `git log` para obtener la fecha de la última confirmación de

cambios, y sustituye con ella todas las cadenas $Date$ que encuentre en el flujo de entrada stdin; imprimiendo luego los resultados. Debería de ser sencillo de implementarlo en cualquier otro lenguaje que domines.

Puedes llamar expanddate a este archivo y ponerlo en el path de ejecución. Tras ello, has de poner un filtro en Git (podemos llamarle dater), e indicarle que use el filtro expanddate para manchar (smudge) los archivos al extraerlos (checkout). Puedes utilizar una expresión Perl para limpiarlos (clean) al almacenarlos (commit):

```
$ git config filter.dater.smudge expand_date
$ git config filter.dater.clean 'perl -pe "s/\\\$Date[^\\\$]*\\\$/\\\$Date\\\$/"'
```

Esta expresión Perl extrae cualquier cosa que vea dentro de una cadena $Date$, para devolverla a como era en un principio. Una vez preparado el filtro, puedes comprobar su funcionamiento preparando un archivo que contenga la clave $Date$ e indicando a Git cual es el atributo para reconocer ese tipo de archivo:

```
$ echo '# $Date$' > date_test.txt
$ echo 'date*.txt filter=dater' >> .gitattributes
```

Al confirmar cambios (commit) y luego extraer (checkout) el archivo de vuelta, verás la clave sustituida:

```
$ git add date_test.txt .gitattributes
$ git commit -m "Testing date expansion in Git"
$ rm date_test.txt
$ git checkout date_test.txt
$ cat date_test.txt
# $Date: Tue Apr 21 07:26:52 2009 -0700$
```

Esta es una muestra de lo poderosa que puede resultar esta técnica para aplicaciones personalizadas. No obstante, debes de ser cuidadoso, ya que el archivo .gitattibutes se almacena y se transmite junto con el proyecto; pero no así el propio filtro, (en este caso, dater), sin el cual no puede funcionar. Cuando diseñes este tipo de filtros, han de estar pensados para que el proyecto continúe funcionando correctamente incluso cuando fallen.

Exportación del repositorio

Los atributos de Git permiten realizar algunas cosas interesantes cuando exportas un archivo de tu proyecto.

export-ignore

Puedes indicar a Git que ignore y no exporte ciertos archivos o carpetas cuando genera un archivo de almacenamiento. Cuando tienes alguna carpeta o archivo que no deseas incluir en tus registros, pero quieras tener controlado en tu proyecto, puedes marcarlos a través del atributo export-ignore.

Por ejemplo, supongamos que tienes algunos archivos de pruebas en la carpeta `test/`, y que no tiene sentido incluirlos en los archivos comprimidos (tarball) al exportar tu proyecto. Puedes añadir la siguiente línea al archivo de atributos de Git:

```
test/ export-ignore
```

A partir de ese momento, cada vez que lances el comando `git archive` para crear un archivo comprimido de tu proyecto, esa carpeta no se incluirá en él.

`export-subst`

Otra cosa que puedes realizar sobre tus archivos es algún tipo de sustitución simple de claves. Git te permite poner la cadena `$Format:$` en cualquier archivo, con cualquiera de las claves de formateo de `--pretty=format` que vimos en el capítulo 2. Por ejemplo, si deseas incluir un archivo llamado `LAST_COMMIT` en tu proyecto, y poner en él automáticamente la fecha de la última confirmación de cambios cada vez que lances el comando `git archive`:

```
$ echo 'Last commit date: $Format:%cd$' > LAST_COMMIT
$ echo "LAST_COMMIT export-subst" >> .gitattributes
$ git add LAST_COMMIT .gitattributes
$ git commit -am 'adding LAST_COMMIT file for archives'
```

Cuando lances la orden `git archive`, lo que la gente verá en ese archivo cuando lo abra será:

```
$ cat LAST_COMMIT
Last commit date: $Format:Tue Apr 21 08:38:48 2009 -0700$
```

Estrategias de fusión (merge)

También puedes utilizar los atributos Git para indicar distintas estrategias de fusión para archivos específicos de tu proyecto. Una opción muy útil es la que nos permite indicar a Git que no intente fusionar ciertos archivos concretos cuando tengan conflictos, manteniendo en su lugar tus archivos sobre los de cualquier otro.

Puede ser interesante si una rama de tu proyecto es divergente o esta especializada, pero deseas seguir siendo capaz de fusionar cambios de vuelta desde ella, e ignorar ciertos archivos. Digamos que tienes un archivo de datos denominado `database.xml`, distinto en las dos ramas, y que deseas fusionar en la otra rama sin perturbarlo. Puedes ajustar un atributo tal como:

```
database.xml merge=ours
```

Y luego definir una estrategia `ours` con:

```
$ git config --global merge.ours.driver true
```

Si fusionas en la otra rama, en lugar de tener conflictos de fusión con el archivo `database.xml`, verás algo como:

```
$ git merge topic
Auto-merging database.xml
Merge made by recursive.
```

Y el archivo `database.xml` permanecerá inalterado en cualquier que fuera la versión que tenias originalmente.

Puntos de enganche en Git

Al igual que en otros sistemas de control de versiones, Git también cuenta con mecanismos para lanzar scripts de usuario cuando suceden ciertas acciones importantes, llamados puntos de enganche (hooks). Hay dos grupos de esos puntos de lanzamiento: los del lado cliente y los del lado servidor. Los puntos del lado cliente están relacionados con operaciones tales como la confirmación de cambios (commit) o la fusión (merge). Los del lado servidor están relacionados con operaciones tales como la recepción de contenidos enviados (push) a un servidor. Estos puntos de enganche pueden utilizarse para multitud de aplicaciones. Vamos a ver unas pocas de ellas.

Instalación de un punto de enganche

Los puntos de enganche se guardan en la subcarpeta *hooks* de la carpeta Git. En la mayoría de proyectos, estará en `.git/hooks`. Por defecto, esta carpeta contiene unos cuantos scripts de ejemplo. Algunos de ellos son útiles por sí mismos; pero su misión principal es la de documentar las variables de entrada para cada script. Todos los ejemplos se han escrito como scripts de shell, con algo de código Perl embebido en ellos. Pero cualquier tipo de script ejecutable que tenga el nombre adecuado puede servir igual de bien. Los puedes escribir en Ruby o en Python o en cualquier lenguaje de scripting con el que trabajes. Si quieres usar los ejemplos que trae Git, tendrás que renombrarlos, ya que los ejemplos acaban su nombre en `.sample`.

Para activar un punto de enganche para un script, pon el archivo correspondiente en la carpeta `hooks`; con el nombre adecuado y con la marca de ejecutable. A partir de ese momento, será automáticamente lanzado cuando se dé la acción correspondiente. Vamos a ver la mayoría de nombres de puntos de enganche disponibles.

Puntos de enganche del lado cliente

Hay muchos de ellos. En esta sección los dividiremos en puntos de enganche en el flujo de trabajo de confirmación de cambios, puntos en el flujo de trabajo de correo electrónico y todos los demás.

NOTE Observa que los puntos de enganche del lado del cliente **no se copian** cuando clonas el repositorio. Si quieres que tengan un efecto para forzar una política es necesario que esté en el lado del cliente. Por ejemplo, mira en Un ejemplo de implantación de una determinada política en Git.

Puntos en el flujo de trabajo de confirmación de cambios

Los primeros cuatro puntos de enganche están relacionados con el proceso de confirmación de cambios.

Primero se activa el punto de enganche `pre-commit`, incluso antes de que teclees el mensaje de confirmación. Se suele utilizar para inspeccionar la instantánea (snapshot) que vas a confirmar, para ver si has olvidado algo, para asegurar que las pruebas se ejecutan, o para revisar cualquier aspecto que necesites inspeccionar en el código. Saliendo con un valor de retorno distinto de cero, se aborta la confirmación de cambios. Aunque siempre puedes saltártelo con la orden `git commit --no-verify`. Puede ser útil para realizar tareas tales como revisar el estilo del código (lanzando `lint` o algo equivalente), revisar los espacios en blanco de relleno (el script de ejemplo hace exactamente eso), o revisar si todos los nuevos métodos llevan la adecuada documentación.

El punto de enganche `prepare-commit-msg` se activa antes de arrancar el editor del mensaje de confirmación de cambios, pero después de crearse el mensaje por defecto. Te permite editar el mensaje por defecto, antes de que lo vea el autor de la confirmación de cambios. Este punto de enganche recibe varias entradas: la ubicación (path) del archivo temporal donde se almacena el mensaje de confirmación, el tipo de confirmación y la clave SHA-1 si estamos enmendando un commit existente. Este punto de enganche no tiene mucha utilidad para las confirmaciones de cambios normales; pero sí para las confirmaciones donde el mensaje por defecto es autogenerado, como en las confirmaciones de fusiones (merge), los mensajes con plantilla, las confirmaciones aplastadas (squash), o las confirmaciones de corrección (amend). Se puede utilizar combinándolo con una plantilla de confirmación, para poder insertar información automáticamente.

El punto de enganche `commit-msg` recibe un parámetro: la ubicación (path) del archivo temporal que contiene el mensaje de confirmación actual. Si este script termina con un código de salida distinto de cero, Git aborta el proceso de confirmación de cambios; permitiendo así validar el estado del proyecto o el mensaje de confirmación antes de permitir continuar. En la última parte de este capítulo, veremos cómo podemos utilizar este punto de enganche para revisar si el mensaje de confirmación es conforme a un determinado patrón obligatorio.

Después de completar todo el proceso de confirmación de cambios, es cuando se lanza el punto de enganche `post-commit`. Este no recibe ningún parámetro, pero podemos obtener fácilmente la última confirmación de cambios con el comando `git log -1 HEAD`. Habitualmente, este script final se suele utilizar para realizar notificaciones o tareas similares.

Puntos en el flujo de trabajo del correo electrónico

Tienes disponibles tres puntos de enganche en el lado cliente para interactuar con el flujo de trabajo de correo electrónico. Todos ellos se invocan al utilizar el comando `git am`, por lo que si no utilizas dicho comando, puedes saltar directamente a la siguiente sección. Si recibes parches a través de correo-e preparados con `git format-patch`, es posible que parte de lo descrito en esta sección te pueda ser útil.

El primer punto de enganche que se activa es `applypatch-msg`. Recibe un solo argumento: el nombre del archivo temporal que contiene el mensaje de confirmación propuesto. Git abortará la aplicación del parche si este script termina con un código de salida distinto de cero. Puedes utilizarlo para asegurarte de que el mensaje de confirmación esté correctamente formateado o para normalizar el mensaje permitiendo al script que lo edite sobre la marcha.

El siguiente punto de enganche que se activa al aplicar parches con `git am` es el punto `pre-applypatch`. No recibe ningún argumento de entrada y se lanza después de que el parche haya sido aplicado, por lo que puedes utilizarlo para revisar la situación (snapshot) antes de confirmarla. Con este script, puedes lanzar pruebas o similares para chequear el árbol de trabajo. Si falta algo o si alguna de las pruebas falla, saliendo con un código de salida distinto de cero abortará el comando `git am` sin confirmar el parche.

El último punto de enganche que se activa durante una operación `git am` es el punto `post-applypatch`. Puedes utilizarlo para notificar de su aplicación al grupo o al autor del parche. No puedes detener el proceso de parcheo con este script.

Otros puntos de enganche del lado cliente

El punto `pre-rebase` se activa antes de cualquier reorganización y puede abortarla si retorna con un código de salida distinto de cero. Puedes usarlo para impedir reorganizaciones de cualquier confirmación de cambios ya enviada (push) a algún servidor. El script de ejemplo para `pre-rebase` hace precisamente eso, aunque asumiendo que `next` es el nombre de la rama publicada. Si lo vas a utilizar, tendrás que modificarlo para que se ajuste al nombre que tenga tu rama publicada.

El punto de enganche `post-rewrite` se ejecuta con los comandos que reemplazan confirmaciones de cambio, como `git commit --amend` y `git rebase` (pero no con `git filter-branch`). Su único argumento es el comando que disparará la reescritura, y recibe una lista de reescrituras por la entrada estándar (`stdin`). Este enganche tiene muchos usos similares a los puntos `post-checkout` y `post-merge`.

Tras completarse la ejecución de un comando `git checkout`, es cuando se activa el punto de enganche `post-checkout`. Lo puedes utilizar para ajustar tu carpeta de trabajo al entorno de tu proyecto. Entre otras cosas, puedes mover grandes archivos binarios de los que no quieras llevar control, puedes autogenerar documentación, y otras cosas.

El punto de enganche `post-merge` se activa tras completarse la ejecución de un comando `git merge`. Puedes utilizarlo para recuperar datos de tu carpeta de trabajo que Git no puede controlar, como por ejemplo datos relativos a permisos. Este punto de enganche puede utilizarse también para comprobar la presencia de ciertos archivos, externos al control de Git, que desees copiar cada vez que cambie la carpeta de trabajo.

El punto `pre-push` se ejecuta durante un `git push`, justo cuando las referencias remotas se han actualizado, pero antes de que los objetos se transfieran. Recibe como parámetros el nombre y la localización del remoto, y una lista de referencias para ser actualizadas, a través de la entrada estándar. Puedes utilizarlo para validar un conjunto de actualizaciones de referencias antes de que la operación de `push` tenga lugar (ya que si el script retorna un valor distinto de cero, se abortará la operación).

En ocasiones, Git realizará una recolección de basura como parte de su funcionamiento habitual, llamando a `git gc --auto`. El punto de enganche `pre-auto-gc` es el que se llama justo antes de realizar dicha recolección de basura, y puede utilizarse para notificarte que tiene lugar dicha operación, o para poderla abortar si se considera que no es un buen momento.

Puntos de enganche del lado servidor

Aparte de los puntos del lado cliente, como administrador de sistemas, puedes utilizar un par de puntos de enganche importantes en el lado servidor; para implementar prácticamente cualquier tipo de política que quieras mantener en tu proyecto. Estos scripts se lanzan antes y después de cada envío (push) al servidor. El script previo, puede terminar con un código de salida distinto de cero y abortar el envío, devolviendo el correspondiente mensaje de error al cliente. Este script puede implementar políticas de recepción tan complejas como desees.

`pre-receive`

El primer script que se activa al manejar un envío de un cliente es el correspondiente al punto de enganche `pre-receive`. Recibe una lista de referencias que se están enviando (push) desde la entrada estándar (stdin); y, si termina con un código de salida distinto de cero, ninguna de ellas será aceptada. Puedes utilizar este punto de enganche para realizar tareas tales como la de comprobar que ninguna de las referencias actualizadas no son de avance directo (non-fast-forward); o para comprobar que el usuario que realiza el envío tiene realmente permisos para para crear, borrar o modificar cualquiera de los archivos que está tratando de cambiar.

`update`

El punto de enganche `update` es muy similar a `pre-receive`, pero con la diferencia de que se activa una vez por cada rama que se está intentando actualizar con el envío. Si la persona que realiza el envío intenta actualizar varias ramas, `pre-receive` se ejecuta una sola vez, mientras que `update` se ejecuta tantas veces como ramas se estén actualizando. El lugar de recibir datos desde la entrada estándar (stdin), este script recibe tres argumentos: el nombre de la rama, la clave SHA-1 a la que esta apuntada antes del envío, y la clave SHA-1 que el usuario está intentando enviar. Si el script `update` termina con un código de salida distinto de cero, únicamente los cambios de esa rama son rechazados; el resto de ramas continuarán con sus actualizaciones.

`post-receive`

El punto de enganche `post-receive` se activa cuando termina todo el proceso, y se puede utilizar para actualizar otros servicios o para enviar notificaciones a otros usuarios. Recibe los mismos datos que `pre-receive` desde la entrada estándar. Algunos ejemplos de posibles aplicaciones pueden ser la de alimentar una lista de correo-e, avisar a un servidor de integración continua, o actualizar un sistema de seguimiento de tickets de servicio (pudiendo incluso procesar el mensaje de confirmación para ver si hemos de abrir, modificar o dar por cerrado algún ticket). Este script no puede detener el proceso de envío, pero el cliente no se desconecta hasta que no se completa su ejecución; por tanto, has de ser cuidadoso cuando intentes realizar con él tareas que puedan requerir mucho tiempo.

Un ejemplo de implantación de una determinada política en Git

En esta sección, utilizarás lo aprendido para establecer un flujo de trabajo en Git que: compruebe si los mensajes de confirmación de cambios encajan en un determinado formato, obligue a realizar solo envíos de avance directo, y permita solo a ciertos usuarios modificar ciertas carpetas del proyecto. Para ello, has de preparar los correspondientes scripts de cliente (para ayudar a los

desarrolladores a saber de antemano si sus envíos van a ser rechazados o no), y los correspondientes scripts de servidor (para obligar a cumplir esas políticas).

He usado Ruby para escribir los ejemplos, tanto porque es mi lenguaje preferido de scripting y porque creo que es el más parecido a pseudocódigo; de tal forma que puedas ser capaz de seguir el código, incluso si no conoces Ruby. Pero, puede ser igualmente válido cualquier otro lenguaje. Todos los script de ejemplo que vienen de serie con Git están escritos en Perl o en Bash shell, por lo que tienes bastantes ejemplos en esos lenguajes de scripting.

Punto de enganche en el lado servidor

Todo el trabajo del lado servidor va en el script `update` de la carpeta `hooks`. Dicho script se lanza cada vez que alguien sube algo a alguna rama, y tiene tres argumentos:

- El nombre de la referencia que se está subiendo
- La vieja revisión de la rama que se está modificando
- La nueva revisión que se está subiendo a la rama

También puedes tener acceso al usuario que está enviando, si este los envía a través de SSH. Si has permitido a cualquiera conectarse con un mismo usuario (como "git", por ejemplo), has tenido que dar a dicho usuario una envoltura (shell wraper) que te permite determinar cuál es el usuario que se conecta según sea su clave pública, permitiéndote fijar una variable de entorno especificando dicho usuario. Aquí, asumiremos que el usuario conectado queda reflejado en la variable de entorno `$USER`, de tal forma que el script `update` comienza recogiendo toda la información que necesitas:

```
#!/usr/bin/env ruby

$refname = ARGV[0]
$oldrev  = ARGV[1]
$newrev  = ARGV[2]
$user    = ENV['USER']

puts "Enforcing Policies..."
puts "(#{$refname}) (#{$oldrev[0,6]}) (#{$newrev[0,6]})"
```

Sí, estoy usando variables globales. No me juzgues por ello, que es más sencillo mostrarlo de esta manera.

Obligando a utilizar un formato específico en el mensaje de commit

Tu primer desafío es asegurarte que todos y cada uno de los mensajes de confirmación de cambios se ajustan a un determinado formato. Simplemente, por fijar algo concreto, supongamos que cada mensaje ha de incluir un texto tal como "ref: 1234", porque quieres enlazar cada confirmación de cambios con una determinada entrada de trabajo en un sistema de control. Has de mirar en cada confirmación de cambios (commit) recibida, para ver si contiene ese texto; y, si no lo trae, salir con un código distinto de cero, de tal forma que el envío (push) sea rechazado.

Puedes obtener la lista de las claves SHA-1 de todos las confirmaciones de cambios enviadas cogiendo los valores de `$newrev` y de `$oldrev`, y pasándolos a comando de mantenimiento de Git llamado `git rev-list`. Este comando es básicamente el mismo que `git log`, pero por defecto, imprime solo los valores SHA-1 y nada más. Con él, puedes obtener la lista de todas las claves SHA que se han introducido entre una clave SHA y otra clave SHA dadas; obtendrás algo así como esto:

```
$ git rev-list 538c33..d14fc7
d14fc7c847ab946ec39590d87783c69b031bdfb7
9f585da4401b0a3999e84113824d15245c13f0be
234071a1be950e2a8d078e6141f5cd20c1e61ad3
dfa04c9ef3d5197182f13fb5b9b1fb7717d2222a
17716ec0f1ff5c77eff40b7fe912f9f6cfd0e475
```

Puedes coger esta salida, establecer un bucle para recorrer cada una de esas confirmaciones de cambios, coger el mensaje de cada una y comprobarlo contra una expresión regular de búsqueda del patrón deseado.

Tienes que imaginarte cómo puedes obtener el mensaje de cada una de esas confirmaciones de cambios a comprobar. Para obtener los datos "en crudo" de una confirmación de cambios, puedes utilizar otro comando de mantenimiento de Git denominado `git cat-file`. En Los entresijos internos de Git volveremos en detalle sobre estos comandos de mantenimiento; pero, por ahora, esto es lo que obtienes con dicho comando:

```
$ git cat-file commit ca82a6
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

changed the version number
```

Una vía sencilla para obtener el mensaje, es la de ir hasta la primera línea en blanco y luego coger todo lo que siga a esta. En los sistemas Unix, lo puedes realizar con el comando `sed`:

```
$ git cat-file commit ca82a6 | sed '1,/^$/d'
changed the version number
```

Puedes usar este "truco de magia" para coger el mensaje de cada confirmación de cambios que se está enviando y salir si localizas algo que no cuadra en alguno de ellos. Para salir del script y rechazar el envío, recuerda que debes salir con un código distinto de cero. El método completo será algo así como:

```
$regex = /\[ref: (\d+)\]/

# enforced custom commit message format
def check_message_format
  missed_revs = `git rev-list #{$oldrev}..#{$newrev}`.split("\n")
  missed_revs.each do |rev|
    message = `git cat-file commit #{rev} | sed '1,/^$/d'`
    if !$regex.match(message)
      puts "[POLICY] Your message is not formatted correctly"
      exit 1
    end
  end
end
check_message_format
```

Poniendo esto en tu script `update`, serán rechazadas todas las actualizaciones que contengan cambios con mensajes que no se ajusten a tus reglas.

Implementando un sistema de listas de control de acceso (ACL)

Imaginemos que deseas implementar un sistema de control de accesos (Access Control List, ACL), para vigilar qué usuarios pueden enviar (push) cambios a qué partes de tus proyectos. Algunas personas tendrán acceso completo, y otras tan solo acceso a ciertas carpetas o a ciertos archivos. Para implementar esto, has de escribir esas reglas de acceso en un archivo denominado `acl` ubicado en tu repositorio git básico (bare) en el servidor. Y tienes que preparar el enganche `update` para hacerle consultar esas reglas, mirar los archivos que están siendo subidos en las confirmaciones de cambio (commit) enviadas (push), y determinar así si el usuario emisor del envío tiene o no permiso para actualizar esos archivos.

El primer paso es escribir tu lista de control de accesos (ACL). Su formato es muy parecido al del mecanismo ACL de CVS: utiliza una serie de líneas donde el primer campo es `avail` o `unavail` (permitido o no permitido), el segundo campo es una lista de usuarios separados por comas, y el último campo es la ubicación (path) sobre el que aplicar la regla (dejarlo en blanco equivale a un acceso abierto). Cada uno de esos campos se separan entre sí con el carácter barra vertical (`|`).

Por ejemplo, si tienes un par de administradores, algunos redactores técnicos con acceso a la carpeta `doc`, y un desarrollador que únicamente accede a las carpetas `lib` y `tests`, el archivo ACL resultante sería:

```
avail|nickh,pjhyett,defunkt,tpw
avail|usinclair,cdickens,ebronte|doc
avail|schacon|lib
avail|schacon|tests
```

Para implementarlo, hemos de leer previamente estos datos en una estructura que podamos emplear. En este caso, por razones de simplicidad, vamos a mostrar únicamente la forma de implementar las directivas `avail` (permitir). Este es un método que te devuelve un array asociativo cuya clave es el nombre del usuario y su valor es un array de ubicaciones (paths) donde ese usuario

tiene acceso de escritura:

```
def get_acl_access_data(acl_file)
  # read in ACL data
  acl_file = File.read(acl_file).split("\n").reject { |line| line == '' }
  access = {}
  acl_file.each do |line|
    avail, users, path = line.split('|')
    next unless avail == 'avail'
    users.split(',').each do |user|
      access[user] ||= []
      access[user] << path
    end
  end
  access
end
```

Si lo aplicamos sobre la lista ACL descrita anteriormente, este método `get_acl_access_data` devolverá una estructura de datos similar a esta:

```
{"defunkt"=>[nil],
 "tpw"=>[nil],
 "nickh"=>[nil],
 "pjhyett"=>[nil],
 "schacon"=>["lib", "tests"],
 "cdickens"=>["doc"],
 "usinclair"=>["doc"],
 "ebronte"=>["doc"]}
```

Una vez tienes los permisos en orden, necesitas averiguar las ubicaciones modificadas por las confirmaciones de cambios enviadas; de tal forma que puedas asegurarte de que el usuario que las está enviando tiene realmente permiso para modificarlas.

Puedes comprobar fácilmente qué archivos han sido modificados en cada confirmación de cambios, utilizando la opción `--name-only` del comando `git log` (citado brevemente en el capítulo 2):

```
$ git log -1 --name-only --pretty=format:'' 9f585d

README
lib/test.rb
```

Utilizando la estructura ACL devuelta por el método `get_acl_access_data` y comprobándola sobre la lista de archivos de cada confirmación de cambios, puedes determinar si el usuario tiene o no permiso para enviar dichos cambios:

```
 # only allows certain users to modify certain subdirectories in a project
def check_directory_perms
  access = get_acl_access_data('acl')

  # see if anyone is trying to push something they can't
  new_commits = `git rev-list #{$oldrev}..#{$newrev}`.split("\n")
  new_commits.each do |rev|
    files_modified = `git log -1 --name-only --pretty=format:'' #{rev}`.split("\n")
    files_modified.each do |path|
      next if path.size == 0
      has_file_access = false
      access[$user].each do |access_path|
        if !access_path  # user has access to everything
           || (path.start_with? access_path) # access to this path
          has_file_access = true
        end
      end
      if !has_file_access
        puts "[POLICY] You do not have access to push to #{path}"
        exit 1
      end
    end
  end
end

check_directory_perms
```

Se puede obtener una lista de los nuevos commits a enviar con `git rev-list`. Para cada uno de ellos, puedes ver qué archivos se quieren modificar y asegurarse que el usuario que está enviando los archivos tiene acceso a todos ellos.

Desde este momento, los usuarios ya no podrán subir cambios con mensajes de confirmación que no cumplan las reglas, o cuando intenten modificar archivos a los que no tienen acceso.

Comprobación

Si lanzas `chmod u+x .git/hooks/update`, siendo este el archivo en el que hemos introducido el código anterior, y probamos a subir un commit con un mensaje que no cumple las reglas, verás algo como esto:

```
$ git push -f origin master
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 323 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
Enforcing Policies...
(refs/heads/master) (8338c5) (c5b616)
[POLICY] Your message is not formatted correctly
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

Hay un par de cosas interesantes aquí. Lo primero es que ves dónde empieza la ejecución del enganche.

```
Enforcing Policies...
(refs/heads/master) (fb8c72) (c56860)
```

Recuerda que imprimías esto al comienzo del script. Todo lo que el script imprima sobre la salida estándar (`stdout`) se enviará al cliente.

Lo siguiente que ves es el mensaje de error.

```
[POLICY] Your message is not formatted correctly
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
```

La primera línea la imprime tu script, las otras dos son las que usa Git para decirte que el script finalizó con error (devuelve un valor distinto de cero) y que está rechazando tu envío. Por último, tienes esto:

```
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

Verás un mensaje de rechazo remoto para cada referencia que tu script ha rechazado, y te dice qué fue rechazado explícitamente debido al fallo del script de enganche.

Y más aún, si alguien intenta realizar un envío, en el que haya confirmaciones de cambio que afecten a archivos a los que ese usuario no tiene acceso, verá algo similar. Por ejemplo, si un documentalista intenta tocar algo de la carpeta `lib`, verá esto:

```
[POLICY] You do not have access to push to lib/test.rb
```

Y eso es todo. De ahora en adelante, en tanto en cuando el script `update` esté presente y sea ejecutable, tu repositorio nunca se verá perjudicado, nunca tendrá un mensaje de confirmación de cambios sin tu plantilla y tus usuarios estarán controlados.

Puntos de enganche del lado cliente

Lo malo del sistema descrito en la sección anterior pueden ser los lamentos que inevitablemente se van a producir cuando los envíos de tus usuarios sean rechazados. Ver rechazado en el último minuto su tan cuidadosamente preparado trabajo, puede ser realmente frustrante. Y, aún peor, tener que reescribir su histórico para corregirlo puede ser un auténtico calvario.

La solución a este dilema es el proporcionarles algunos enganches (hook) del lado cliente, para que les avisen cuando están trabajando en algo que el servidor va a rechazarles. De esta forma, pueden corregir los problemas antes de confirmar cambios y antes de que se conviertan en algo realmente complicado de arreglar. Debido a que estos enganches no se transfieren junto con el clonado de un proyecto, tendrás que distribuirlos de alguna otra manera. Y luego pedir a tus usuarios que se los copien a sus carpetas `.git/hooks` y los hagan ejecutables. Puedes distribuir esos enganches dentro del mismo proyecto o en un proyecto separado. Pero no hay modo de implementarlos automáticamente.

Para empezar, se necesita chequear el mensaje de confirmación inmediatamente antes de cada confirmación de cambios, para asegurarse de que el servidor no los rechazará debido a un mensaje mal formateado. Para ello, se añade el enganche `commit-msg`. Comparando el mensaje del archivo pasado como primer argumento con el mensaje patrón, puedes obligar a Git a abortar la confirmación de cambios (commit) en caso de no coincidir ambos:

```
#!/usr/bin/env ruby
message_file = ARGV[0]
message = File.read(message_file)

$regex = /\[ref: (\d+)\]/

if !$regex.match(message)
  puts "[POLICY] Your message is not formatted correctly"
  exit 1
end
```

Si este script está en su sitio (el archivo `.git/hooks/commit-msg`) y es ejecutable, al confirmar cambios con un mensaje inapropiado, verás algo así como:

```
$ git commit -am 'test'
[POLICY] Your message is not formatted correctly
```

Y la confirmación no se llevará a cabo. Sin embargo, si el mensaje está formateado adecuadamente,

Git te permitirá confirmar cambios:

```
$ git commit -am 'test [ref: 132]'
[master e05c914] test [ref: 132]
 1 file changed, 1 insertions(+), 0 deletions(-)
```

A continuación, se necesita también asegurarse de no estar modificando archivos fuera del alcance de tus permisos. Si la carpeta `.git` de tu proyecto contiene una copia del archivo de control de accesos (ACL) utilizada previamente, este script `pre-commit` podrá comprobar los límites:

```
#!/usr/bin/env ruby

$user    = ENV['USER']

 # [ insert acl_access_data method from above ]

 # only allows certain users to modify certain subdirectories in a project
def check_directory_perms
  access = get_acl_access_data('.git/acl')

  files_modified = `git diff-index --cached --name-only HEAD`.split("\n")
  files_modified.each do |path|
    next if path.size == 0
    has_file_access = false
    access[$user].each do |access_path|
    if !access_path || (path.index(access_path) == 0)
      has_file_access = true
    end
    if !has_file_access
      puts "[POLICY] You do not have access to push to #{path}"
      exit 1
    end
  end
end

check_directory_perms
```

Este es un script prácticamente igual al del lado servidor. Pero con dos importantes diferencias. La primera es que el archivo ACL está en otra ubicación, debido a que el script corre desde tu carpeta de trabajo y no desde la carpeta de Git. Esto obliga a cambiar la ubicación del archivo ACL de aquí:

```
access = get_acl_access_data('acl')
```

a este otro sitio:

```
access = get_acl_access_data('.git/acl')
```

La segunda diferencia es la forma de listar los archivos modificados. Debido a que el método del lado servidor utiliza el registro de confirmaciones de cambio, pero, sin embargo, aquí la confirmación no se ha registrado aún, la lista de archivos se ha de obtener desde el área de preparación (staging area). En lugar de:

```
files_modified = `git log -1 --name-only --pretty=format:'' #{ref}`
```

tenemos que utilizar:

```
files_modified = `git diff-index --cached --name-only HEAD`
```

Estas dos son las únicas diferencias; en todo lo demás, el script funciona de la misma manera. Es necesario advertir de que se espera que trabajes localmente con el mismo usuario con el que enviarás (push) a la máquina remota. Si no fuera así, tendrás que ajustar manualmente la variable `$user`.

El último aspecto a comprobar es el de no intentar enviar referencias que no sean de avance-rápido. Pero esto es algo más raro que suceda. Para tener una referencia que no sea de avance-rápido, tienes que haber reorganizado (rebase) una confirmación de cambios (commit) ya enviada anteriormente, o tienes que estar tratando de enviar una rama local distinta sobre la misma rama remota.

De todas formas, el único aspecto accidental que puede interesante capturar son los intentos de reorganizar confirmaciones de cambios ya enviadas. El servidor te avisará de que no puedes enviar ningún no-avance-rapido, y el enganche te impedirá cualquier envío forzado

Este es un ejemplo de script previo a reorganización que lo puede comprobar. Con la lista de confirmaciones de cambio que estás a punto de reescribir, las comprueba por si alguna de ellas existe en alguna de tus referencias remotas. Si encuentra alguna, aborta la reorganización:

```
#!/usr/bin/env ruby

base_branch = ARGV[0]
if ARGV[1]
  topic_branch = ARGV[1]
else
  topic_branch = "HEAD"
end

target_shas = `git rev-list #{base_branch}..#{topic_branch}`.split("\n")
remote_refs = `git branch -r`.split("\n").map { |r| r.strip }

target_shas.each do |sha|
  remote_refs.each do |remote_ref|
    shas_pushed = `git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
    if shas_pushed.split("\n").include?(sha)
      puts "[POLICY] Commit #{sha} has already been pushed to #{remote_ref}"
      exit 1
    end
  end
end
```

Este script utiliza una sintaxis no contemplada en la sección de Selección de Revisiones del capítulo 6. La lista de confirmaciones de cambio previamente enviadas, se comprueba con:

```
`git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
```

La sintaxis *SHA^@* recupera todos los padres de esa confirmación de cambios (commit). Estás mirando por cualquier confirmación que se pueda alcanzar desde la última en la parte remota, pero que no se pueda alcanzar desde ninguno de los padres de cualquiera de las claves SHA que estás intentando enviar. Es decir, confirmaciones de avance-rápido.

La mayor pega de este sistema es el que puede llegar a ser muy lento; y muchas veces es innecesario, ya que el propio servidor te va a avisar y te impedirá el envío, siempre y cuando no intentes forzar dicho envío con la opción *-f*. De todas formas, es un ejercicio interesante. Y, en teoría al menos, pude ayudarte a evitar reorganizaciones que luego tengas de echar para atrás y arreglarlas.

Recapitulación

Se han visto las principales vías por donde puedes personalizar tanto tu cliente como tu servidor Git para que se ajusten a tu forma de trabajar y a tus proyectos. Has aprendido todo tipo de ajustes de configuración, atributos basados en archivos e incluso enganches (hooks). Y has preparado un ejemplo de servidor con mecanismos para asegurar políticas determinadas. A partir de ahora estás listo para encajar Git en prácticamente cualquier flujo de trabajo que puedas imaginar.

Git y Otros Sistemas

El mundo no es perfecto. Por lo general, no se puede cambiar inmediatamente cada proyecto con el que está en contacto Git. A veces estás atrapado en un proyecto usando otro VCS, y desearías poder usar Git. Pasaremos la primera parte de este capítulo aprendiendo sobre cómo utilizar Git como cliente cuando el proyecto en el que se está trabajando está alojado en un sistema diferente.

En algún momento, puede que desees convertir tu proyecto existente a Git. La segunda parte de este capítulo describe cómo migrar tu proyecto en Git desde varios sistemas específicos, así como un método que funcionará si no existe una herramienta de importación pre-construida.

Git como Cliente

Git proporciona una experiencia tan agradable para los desarrolladores que muchas personas han descubierto cómo usarlo en su estación de trabajo, incluso si el resto de su equipo está usando un VCS completamente diferente. Hay un número de estos adaptadores disponibles, llamados "bridges". Aquí vamos a cubrir los que es más probable que se encuentren en la naturaleza.

Git y Subversión

Una gran parte de los proyectos de desarrollo de código abierto y un gran número de proyectos corporativos usan Subversion para administrar su código fuente. Ha existido por más de una década, y la gran parte de ese tiempo fue la elección *de facto* de VCS para proyectos de código abierto. También es similar en muchos aspectos a CVS, que fué el sistema mas popular de control de código del mundo antes de eso.

Una de las mejores características de Git es un puente birideccional para Subversion llamado `git svn`. Esta herramienta te permite usar Git como un cliente válido para un servidor de Subversion, por lo que puedes usar todas las características locales de Git y enviarlo a un servidor de Subversion como si estuvieras usando Subversion localmente. Esto significa que puedes realizar bifurcaciones y fusiones locales, usar el área de preparación, usar el rebasamiento y la selección selectiva, y demás, mientras tus colaboradores continuan trabajando en sus formas oscuras y antiguas. Es una buena forma de introducir Git en el ambiente corporativo y ayudar a tus compañeros desarrolladores a ser más eficientes mientras presionas para que se modifique la infraestructura para que sea compatible con Git por completo. El puente de subversión es la droga de entrada al mundo de DVCS.

`git svn`

El comando base en Git para todos los comandos de puente de Subversion es `git svn`. Se requieren bastantes comandos, por lo que mostraremos los más comunes mientras realizamos algunos flujos de trabajo simples.

Es importante tener en cuenta que cuando usas `git svn`, estás interactuando con Subversion, que es un sistema que funciona de manera muy diferente a Git. Aunque * puede * realizar bifurcaciones y fusiones locales, generalmente es mejor mantener su historial lo más lineal posible al volver a basar su trabajo y evitar hacer cosas como interactuar simultáneamente con un repositorio remoto de Git.

No reescriba su historial y trate de volver a presionar, y no presione a un repositorio paralelo de Git para colaborar con otros desarrolladores de Git al mismo tiempo. Subversion solo puede tener un único historial lineal, y confundirlo es muy fácil. Si trabajas con un equipo, y algunos utilizan SVN y otros usan Git, asegúrate de que todos estén usando el servidor SVN para colaborar, ya que hacerlo te hará la vida más sencilla.

Configurando

Para demostrar esta funcionalidad, necesitas un repositorio SVN típico al que tengas acceso de escritura. Si desea copiar estos ejemplos, tendrá que hacer una copia escribible de mi repositorio de prueba. Para hacer eso fácilmente, puedes usar una herramienta llamada `svnsync` que viene con Subversion.

Para seguir, primero necesitas crear un nuevo repositorio local de Subversion:

```
$ mkdir /tmp/test-svn
$ svnadmin create /tmp/test-svn
```

Luego, habilite a todos los usuarios a cambiar revprops – la manera más facil es añadir un script `pre-revprop-change` que siempre muestra 0:

```
$ cat /tmp/test-svn/hooks/pre-revprop-change
#!/bin/sh
exit 0;
$ chmod +x /tmp/test-svn/hooks/pre-revprop-change
```

Ahora puede sincronizar este proyecto con su máquina local llamando a `svnsync init` con los repositorios

```
$ svnsync init file:///tmp/test-svn \
  http://progit-example.googlecode.com/svn/
```

Esto configura las propiedades para ejecutar la sincronización. A continuación, puede clonar el código ejecutando

```
$ svnsync sync file:///tmp/test-svn
Committed revision 1.
Copied properties for revision 1.
Transmitting file data .............................[...]
Committed revision 2.
Copied properties for revision 2.
[…]
```

A pesar de que esta operación puede tardar solo unos minutos, si intentas copiar el repositorio original a otro repositorio remoto en vez de uno local, el proceso tardará cerca de una hora, aunque

haya menos de 100 commits. Subversion tiene que clonar una revisión a la vez y luego ponerlas en otro repositorio – es ridículamente ineficiente, pero es la única forma fácil de hacerlo.

Empezando

Ahora que tienes un repositorio de Subversion al que tienes acceso para escribir, puedes ir a través de un flujo de trabajo típico. Comenzarás con el comando `git svn clone`, que importa un repositorio completo de Subversion en un repositorio local de Git. Recuerde que si está importando desde un repositorio de Subversion alojado real, debe reemplazar el `file: /// tmp / test-svn` aquí con la URL de su repositorio de Subversion:

```
$ git svn clone file:///tmp/test-svn -T trunk -b branches -t tags
Initialized empty Git repository in /private/tmp/progit/test-svn/.git/
r1 = dcbfb5891860124cc2e8cc616cded42624897125 (refs/remotes/origin/trunk)
    A   m4/acx_pthread.m4
    A   m4/stl_hash.m4
    A   java/src/test/java/com/google/protobuf/UnknownFieldSetTest.java
    A   java/src/test/java/com/google/protobuf/WireFormatTest.java
…
r75 = 556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae (refs/remotes/origin/trunk)
Found possible branch point: file:///tmp/test-svn/trunk => file:///tmp/test-
svn/branches/my-calc-branch, 75
Found branch parent: (refs/remotes/origin/my-calc-branch)
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae
Following parent with do_switch
Successfully followed parent
r76 = 0fb585761df569eaecd8146c71e58d70147460a2 (refs/remotes/origin/my-calc-branch)
Checked out HEAD:
  file:///tmp/test-svn/trunk r75
```

Esto ejecuta el equivalente de dos comandos - `git svn init` seguido de` git svn fetch` - en la URL que proporcione. Esto puede tomar un rato. El proyecto de prueba solo tiene alrededor de 75 confirmaciones y la base de código no es tan grande, pero Git tiene que verificar cada versión, una a la vez, y enviarla por separado. Para un proyecto con cientos o miles de confirmaciones, esto puede demorar literalmente horas o incluso días.

La parte `-T trunk -b branches -t tags` le dice a Git que este repositorio de Subversion sigue las convenciones básicas de bifurcación y etiquetado. Si nombra el tronco, las ramas o las etiquetas de manera diferente, puede cambiar estas opciones. Debido a que esto es tan común, puede reemplazar esta parte completa con `-s`, lo que significa diseño estándar e implica todas esas opciones. El siguiente comando es equivalente:

```
$ git svn clone file:///tmp/test-svn -s
```

En este punto, debe tener un repositorio de Git válido que haya importado sus ramas y etiquetas:

```
$ git branch -a
* master
  remotes/origin/my-calc-branch
  remotes/origin/tags/2.0.2
  remotes/origin/tags/release-2.0.1
  remotes/origin/tags/release-2.0.2
  remotes/origin/tags/release-2.0.2rc1
  remotes/origin/trunk
```

Tenga en cuenta cómo esta herramienta gestiona las etiquetas de Subversion como referencias remotas. Echemos un vistazo más de cerca con el comando de plomería de Git `show-ref`:

```
$ git show-ref
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/heads/master
0fb585761df569eaecd8146c71e58d70147460a2 refs/remotes/origin/my-calc-branch
bfd2d79303166789fc73af4046651a4b35c12f0b refs/remotes/origin/tags/2.0.2
285c2b2e36e467dd4d91c8e3c0c0e1750b3fe8ca refs/remotes/origin/tags/release-2.0.1
cbda99cb45d9abcb9793db1d4f70ae562a969f1e refs/remotes/origin/tags/release-2.0.2
a9f074aa89e826d6f9d30808ce5ae3ffe711feda refs/remotes/origin/tags/release-2.0.2rc1
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/remotes/origin/trunk
```

Git no hace esto cuando se clona desde un servidor Git; esto es lo que parece un repositorio con etiquetas después de un nuevo clon:

```
$ git show-ref
c3dcbe8488c6240392e8a5d7553bbffcb0f94ef0 refs/remotes/origin/master
32ef1d1c7cc8c603ab78416262cc421b80a8c2df refs/remotes/origin/branch-1
75f703a3580a9b81ead89fe1138e6da858c5ba18 refs/remotes/origin/branch-2
23f8588dde934e8f33c263c6d8359b2ae095f863 refs/tags/v0.1.0
7064938bd5e7ef47bfd79a685a62c1e2649e2ce7 refs/tags/v0.2.0
6dcb09b5b57875f334f61aebed695e2e4193db5e refs/tags/v1.0.0
```

Git busca las etiquetas directamente en `refs / tags`, en lugar de tratarlas en sucursales remotas.

Comprometerse con la subversión

Ahora que tiene un repositorio en funcionamiento, puede hacer algo de trabajo en el proyecto e impulsar sus confirmaciones en sentido ascendente, utilizando Git de manera efectiva como un cliente SVN. Si edita uno de los archivos y lo confirma, tiene una confirmación que existe en Git localmente y que no existe en el servidor de Subversion:

```
$ git commit -am 'Adding git-svn instructions to the README'
[master 4af61fd] Adding git-svn instructions to the README
 1 file changed, 5 insertions(+)
```

A continuación, debe impulsar su cambio en sentido ascendente. Observe cómo esto cambia la

forma en que trabaja con Subversion: puede hacer varias confirmaciones fuera de línea y luego enviarlas todas de una vez al servidor de Subversion. Para enviar a un servidor de Subversion, ejecute el comando `git svn dcommit`:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M   README.txt
Committed r77
    M   README.txt
r77 = 95e0222ba6399739834380eb10afcd73e0670bc5 (refs/remotes/origin/trunk)
No changes between 4af61fd05045e07598c553167e0f31c84fd6ffe1 and
refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

Esto toma todas las confirmaciones que hayas realizado sobre el código del servidor de Subversion, una confirmación de Subversion para cada una, y luego reescribe tu compromiso local de Git para incluir un identificador único. Esto es importante porque significa que todas las sumas de comprobación de SHA-1 cambian. En parte por esta razón, trabajar con versiones remotas basadas en Git de sus proyectos simultáneamente con un servidor de Subversion no es una buena idea. Si mira la última confirmación, puede ver el nuevo `git-svn-id` que se agregó:

```
$ git log -1
commit 95e0222ba6399739834380eb10afcd73e0670bc5
Author: ben <ben@0b684db3-b064-4277-89d1-21af03df0a68>
Date:   Thu Jul 24 03:08:36 2014 +0000

    Adding git-svn instructions to the README

    git-svn-id: file:///tmp/test-svn/trunk@77 0b684db3-b064-4277-89d1-21af03df0a68
```

Observe que la suma de comprobación SHA-1 que originalmente comenzó con `4af61fd` cuando usted cometió ahora comienza con` 95e0222`. Si desea enviar tanto a un servidor de Git como a un servidor de Subversion, primero tiene que presionar (*commit*) al servidor de Subversion, porque esa acción cambia sus datos de confirmación.

Tirando hacia nuevos cambios

Si está trabajando con otros desarrolladores, en algún momento uno de ustedes empujará, y luego el otro tratará de impulsar un cambio que entra en conflicto. Ese cambio será rechazado hasta que te combines en su trabajo. En `git svn`, se veeá así:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...

ERROR from SVN:
Transaction is out of date: File '/trunk/README.txt' is out of date
W: d5837c4b461b7c0e018b49d12398769d2bfc240a and refs/remotes/origin/trunk differ,
using rebase:
:100644 100644 f414c433af0fd6734428cf9d2a9fd8ba00ada145
c80b6127dd04f5fcda218730ddf3a2da4eb39138 M  README.txt
Current branch master is up to date.
ERROR: Not all changes have been committed into SVN, however the committed
ones (if any) seem to be successfully integrated into the working tree.
Please see the above messages for details.
```

Para resolver esta situación, puede ejecutar `git svn rebase`, que elimina cualquier cambio en el servidor que aún no tenga y rebases cualquier trabajo que tenga encima de lo que hay en el servidor:

```
$ git svn rebase
Committing to file:///tmp/test-svn/trunk ...

ERROR from SVN:
Transaction is out of date: File '/trunk/README.txt' is out of date
W: eaa029d99f87c5c822c5c29039d19111ff32ef46 and refs/remotes/origin/trunk differ,
using rebase:
:100644 100644 65536c6e30d263495c17d781962cfff12422693a
b34372b25ccf4945fe5658fa381b075045e7702a M  README.txt
First, rewinding head to replay your work on top of it...
Applying: update foo
Using index info to reconstruct a base tree...
M   README.txt
Falling back to patching base and 3-way merge...
Auto-merging README.txt
ERROR: Not all changes have been committed into SVN, however the committed
ones (if any) seem to be successfully integrated into the working tree.
Please see the above messages for details.
```

Ahora, todo su trabajo está encima de lo que hay en el servidor de Subversion, por lo que puede aplicar `dcommit` con éxito:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M   README.txt
Committed r85
    M   README.txt
r85 = 9c29704cc0bbbed7bd58160cfb66cb9191835cd8 (refs/remotes/origin/trunk)
No changes between 5762f56732a958d6cfda681b661d2a239cc53ef5 and
refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

Tenga en cuenta que, a diferencia de Git, que requiere fusionar el trabajo original que todavía no tiene localmente antes de poder enviarlo, `git svn` hace que lo haga solo si los cambios entran en conflicto (muy parecido a cómo funciona Subversion). Si alguien más presiona un cambio en un archivo y luego presiona un cambio en otro archivo, su `dcommit` funcionará bien:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M   configure.ac
Committed r87
    M   autogen.sh
r86 = d8450bab8a77228a644b7dc0e95977ffc61adff7 (refs/remotes/origin/trunk)
    M   configure.ac
r87 = f3653ea40cb4e26b6281cec102e35dcba1fe17c4 (refs/remotes/origin/trunk)
W: a0253d06732169107aa020390d9fefd2b1d92806 and refs/remotes/origin/trunk differ,
using rebase:
:100755 100755 efa5a59965fbbb5b2b0a12890f1b351bb5493c18
e757b59a9439312d80d5d43bb65d4a7d0389ed6d M  autogen.sh
First, rewinding head to replay your work on top of it...
```

Es importante recordar esto, porque el resultado es un estado del proyecto que no existía en ninguna de sus computadoras cuando aplicó Si los cambios son incompatibles pero no conflictivos, es posible que tenga problemas que son difíciles de diagnosticar. Esto es diferente a usar un servidor Git: en Git, puedes probar completamente el estado en tu sistema cliente antes de publicarlo, mientras que en SVN, nunca puedes estar seguro de que los estados inmediatamente antes de la confirmación y después de la confirmación sean idénticos.

También debe ejecutar este comando para obtener cambios del servidor de Subversion, incluso si no está listo para confirmar. Puede ejecutar `git svn fetch` para obtener los nuevos datos, pero` git svn rebase` hace la búsqueda y luego actualiza sus confirmaciones locales.

```
$ git svn rebase
    M   autogen.sh
r88 = c9c5f83c64bd755368784b444bc7a0216cc1e17b (refs/remotes/origin/trunk)
First, rewinding head to replay your work on top of it...
Fast-forwarded master to refs/remotes/origin/trunk.
```

Ejecutar `git svn rebase` de vez en cuando se asegura de que su código esté siempre actualizado.

Debe asegurarse de que su directorio de trabajo esté limpio cuando lo ejecute. Si tiene cambios locales, debe esconder su trabajo o confirmarlo temporalmente antes de ejecutar `git svn rebase`; de lo contrario, el comando se detendrá si ve que la rebase dará como resultado un conflicto de fusión.

Problemas de Git Branching

Cuando te sientas cómodo con un flujo de trabajo de Git, probablemente crearás ramas temáticas, trabajarás en ellas y luego las fusionarás. Si está presionando un servidor de Subversion a través de `git svn`, es posible que desee volver a establecer su trabajo en una sola rama cada vez en lugar de fusionar ramas. La razón para preferir el rebasamiento es que Subversion tiene un historial lineal y no trata con fusiones como Git, por lo que `git svn` sigue al primer padre cuando convierte las instantáneas en confirmaciones de Subversion.

Supongamos que su historial se parece a lo siguiente: creó una rama `experiment`, hizo dos confirmaciones y luego las fusionó de nuevo en `master`. Cuando `dcommit`, ve resultados como este:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M   CHANGES.txt
Committed r89
    M   CHANGES.txt
r89 = 89d492c884ea7c834353563d5d913c6adf933981 (refs/remotes/origin/trunk)
    M   COPYING.txt
    M   INSTALL.txt
Committed r90
    M   INSTALL.txt
    M   COPYING.txt
r90 = cb522197870e61467473391799148f6721bcf9a0 (refs/remotes/origin/trunk)
No changes between 71af502c214ba13123992338569f4669877f55fd and
refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

Ejecutar `dcommit` en una rama con historial combinado funciona bien, excepto que cuando miras el historial de tu proyecto Git, no ha reescrito ninguno de los commits que hiciste en la rama` experimento`; en cambio, todos esos cambios aparecen en el Versión SVN de la confirmación de fusión única.

Cuando alguien más clona ese trabajo, todo lo que ven es la combinación de fusión con todo el trabajo aplastado en ella, como si ejecutaras `git merge --squash`; no ven los datos de confirmación sobre su procedencia o cuándo se cometieron.

Ramificación en Subversion

La ramificación en Subversion no es lo mismo que la bifurcación en Git; si puedes evitar usarlo mucho, probablemente sea lo mejor. Sin embargo, puede crear y comprometerse con ramas en Subversion usando git svn.

Creando una nueva rama de SVN

Para crear una nueva rama en Subversion, ejecuta `git svn branch [branchname]`:

```
$ git svn branch opera
Copying file:///tmp/test-svn/trunk at r90 to file:///tmp/test-svn/branches/opera...
Found possible branch point: file:///tmp/test-svn/trunk => file:///tmp/test-
svn/branches/opera, 90
Found branch parent: (refs/remotes/origin/opera)
cb522197870e61467473391799148f6721bcf9a0
Following parent with do_switch
Successfully followed parent
r91 = f1b64a3855d3c8dd84ee0ef10fa89d27f1584302 (refs/remotes/origin/opera)
```

Esto hace el equivalente del comando `svn copy trunk branches / opera` en Subversion y opera en el servidor de Subversion. Es importante tener en cuenta que no te echa un vistazo en esa rama; si confirma en este punto, esa confirmación irá a `trunk` en el servidor, no en `opera`.

Cambio de ramas activas

Git averigua a qué rama se dirigen tus dcommits buscando la punta de cualquiera de tus ramas de Subversion en tu historial: debes tener solo una, y debería ser la última con un `git-svn-id` en tu rama actual del historial historia.

Si desea trabajar en más de una sucursal al mismo tiempo, puede configurar las sucursales locales para `compromentar` a las sucursales de Subversion específicas, comenzándolas en la confirmación de Subversion importada para esa sucursal. Si desea que una rama `opera ' pueda trabajar por separado, puede ejecutar

```
$ git branch opera remotes/origin/opera
```

Ahora, si desea fusionar su rama *opera* en `trunk` (su rama` master`), puede hacerlo con un `git merge` normal. Pero debe proporcionar un mensaje de compromiso descriptivo (mediante `-m`), o la combinación dirá` 'Merge branch opera '' en lugar de algo útil.

Recuerda que aunque estás usando `git merge` para hacer esta operación la fusión probablemente sea mucho más fácil de lo que sería en Subversion (porque Git detectará automáticamente la base de confirmación apropiada para ti), esto no es una situación de confirmación normal de `git merge` Tienes que volver a enviar estos datos a un servidor de Subversion que no puede manejar una confirmación que rastrea más de un padre; por lo tanto, después de subirlo, se verá como una única confirmación que aplastó todo el trabajo de otra rama en una sola confirmación. Después de fusionar una rama con otra, no puede volver atrás fácilmente y continuar trabajando en esa rama, como lo hace normalmente en Git. El comando `dcommit` que ejecuta borra cualquier información que indique en qué rama se fusionó, por lo que los cálculos subsiguientes de la base de confirmación serán incorrectos: el compromiso hace que el resultado de` git merge` parezca que usted ejecutó `git merge --squash`. Desafortunadamente, no hay una buena forma de evitar esta situación: Subversion no puede almacenar esta información, por lo que siempre estará paralizado por sus limitaciones mientras la usa como su servidor. Para evitar problemas, debe eliminar la

rama local (en este caso, `opera`) después de fusionarla en el enlace troncal.

Comandos de Subversion

El conjunto de herramientas `git svn` proporciona una serie de comandos para ayudar a facilitar la transición a Git al proporcionar una funcionalidad similar a la que tenía en Subversion. Aquí hay algunos comandos que te dan de lo que solía hacer Subversion.

Historial de estilo de SVN

Si está acostumbrado a Subversion y quiere ver su historial en el estilo de salida SVN, puede ejecutar `git svn log` para ver su historial de commit en formato SVN:

```
$ git svn log
------------------------------------------------------------------------
r87 | schacon | 2014-05-02 16:07:37 -0700 (Sat, 02 May 2014) | 2 lines

autogen change

------------------------------------------------------------------------
r86 | schacon | 2014-05-02 16:00:21 -0700 (Sat, 02 May 2014) | 2 lines

Merge branch 'experiment'

------------------------------------------------------------------------
r85 | schacon | 2014-05-02 16:00:09 -0700 (Sat, 02 May 2014) | 2 lines

updated the changelog
```

Debes saber dos cosas importantes sobre `git svn log`. En primer lugar, funciona sin conexión, a diferencia del comando real `svn log`, que solicita al servidor de Subversion los datos. En segundo lugar, solo muestra los compromisos que se han comprometido hasta el servidor de Subversion. Local Git confirma que usted no se ha confirmado a no aparecer; tampoco hay confirmaciones que la gente haya hecho al servidor de Subversion mientras tanto. Es más como el último estado conocido de las confirmaciones en el servidor de Subversion.

Anotación de SVN

Al igual que el comando `git svn log` simula el comando` svn log` fuera de línea, puede obtener el equivalente de `svn annotate` ejecutando` git svn blame [FILE] `. La salida se ve así:

```
$ git svn blame README.txt
 2   temporal Protocol Buffers - Google's data interchange format
 2   temporal Copyright 2008 Google Inc.
 2   temporal http://code.google.com/apis/protocolbuffers/
 2   temporal
22   temporal C++ Installation - Unix
22   temporal =======================
 2   temporal
79    schacon Committing in git-svn.
78    schacon
 2   temporal To build and install the C++ Protocol Buffer runtime and the Protocol
 2   temporal Buffer compiler (protoc) execute the following:
 2   temporal
```

De nuevo, no muestra confirmaciones que hiciste localmente en Git o que se han enviado a Subversion mientras tanto.

Información del servidor SVN

También puede obtener el mismo tipo de información que `svn info` le brinda al ejecutar` git svn info`:

```
$ git svn info
Path: .
URL: https://schacon-test.googlecode.com/svn/trunk
Repository Root: https://schacon-test.googlecode.com/svn
Repository UUID: 4c93b258-373f-11de-be05-5f7a86268029
Revision: 87
Node Kind: directory
Schedule: normal
Last Changed Author: schacon
Last Changed Rev: 87
Last Changed Date: 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009)
```

Esto es como *blame* y *log* ya que se ejecuta fuera de línea y está actualizado solo a partir de la última vez que se comunicó con el servidor de Subversion.

Ignorar lo qué subversión ignora

Si clonas un repositorio de Subversion que tiene propiedades `svn: ignore` establecidas en cualquier lugar, es probable que desees establecer los archivos correspondientes '.gitignore` para que no confirmes accidentalmente los archivos que no deberías. `git svn` tiene dos comandos para ayudar con este problema. El primero es `git svn create-ignore`, que crea automáticamente los archivos correspondientes '.gitignore` para que su próxima confirmacion pueda incluirlos.

El segundo comando es `git svn show-ignore`, que imprime para extender las líneas que necesita poner en un archivo` .gitignore` para que pueda redirigir el resultado al archivo de exclusión de proyecto:

```
$ git svn show-ignore > .git/info/exclude
```

De esta forma, no descarta el proyecto con archivos `.gitignore`. Esta es una buena opción si eres el único usuario de Git en un equipo de Subversion, y tus compañeros de equipo no quieren archivos `.gitignore` en el proyecto.

Resumen de Git-Svn

Las herramientas `git svn` son útiles si estás atascado con un servidor de Subversion, o si no estás en un entorno de desarrollo que necesita ejecutar un servidor de Subversion. Sin embargo, debería considerar que paralizó a Git, o llegará a problemas de traducción que pueden confundirlo a usted y a sus colaboradores. Para evitar problemas, intente seguir estas pautas:

- Mantener un historial lineal de Git que no contenga uniones de fusión hechas por `git merge`. Rebase cualquier trabajo que realice fuera de su rama principal; nos e fusiona.
- No configure y colabore en un servidor Git separado. Posiblemente tenga uno para acelerar clones para nuevos desarrolladores, pero no le empuje nada que no tenga una entrada `git-svn-id`. Incluso puede desear agregar un gancho `pre-receive` que comprueba cada mensaje de confirmación para un` git-svn-id` y rechaza los empujes que contienen commits sin él.

Si sigues esas pautas, trabajar con un servidor de Subversion puede ser más llevadero. Sin embargo, si es posible pasar a un servidor Git real, hacerlo puede hacer que tu equipo gane mucho más.

Git y Mercurial

El universo DVCS es más grande que el de Git. De hecho, hay muchos otros sistemas en este espacio, cada uno con su propio ángulo sobre cómo hacer el control de versión distribuida correctamente. Aparte de Git, el más popular es Mercurial, y los dos son muy similares en muchos aspectos.

La buena noticia, si prefiere el comportamiento del cliente de Git pero que está trabajando con un proyecto cuyo código fuente está controlado con Mercurial, es que hay una manera de usar Git como cliente para un repositorio alojado en Mercurial. Dado que la forma en que Git habla con los repositorios del servidor es a través de controles remotos, no debería sorprendernos que este puente se implemente como un ayudante remoto. El nombre del proyecto es git-remote-hg, y se puede encontrar en https://github.com/felipec/git-remote-hg.

git-remote-hg

Primero, necesitas instalar git-remote-hg. Esto básicamente implica dejar caer su archivo en algún lugar de tu camino, así:

```
$ curl -o ~/bin/git-remote-hg \
  https://raw.githubusercontent.com/felipec/git-remote-hg/master/git-remote-hg
$ chmod +x ~/bin/git-remote-hg
```

i. asumiendo que `~ / bin` está en su` $ PATH`. Git-remote-hg tiene otra dependencia: la biblioteca

mercurial para Python. Si tienes instalado Python, es tan sencillo como:

```
$ pip install mercurial
```

(Si no tiene instalado Python, visite https://www.python.org/ y consígalo primero.)

Lo último que necesitarás es el cliente Mercurial. Vaya a http://mercurial.selenic.com/ e instálelo si aún no lo ha hecho.

Ahora estás listo para el rock. Todo lo que necesitas es un repositorio Mercurial al que puedes presionar. Afortunadamente, todos los repositorios de Mercurial pueden actuar de esta manera, así que solo usaremos el repositorio de "hola mundo" que todos usan para aprender Mercurial:

```
$ hg clone http://selenic.com/repo/hello /tmp/hello
```

Empezando

Ahora que tenemos un repositorio “server-side” adecuado, podemos pasar por un flujo de trabajo típico. Como verá, estos dos sistemas son lo suficientemente similares como para que no haya mucha fricción.

Como siempre con Git, primero clonamos:

```
$ git clone hg::/tmp/hello /tmp/hello-git
$ cd /tmp/hello-git
$ git log --oneline --graph --decorate
* ac7955c (HEAD, origin/master, origin/branches/default, origin/HEAD,
refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master, master) Create a
makefile
* 65bb417 Create a standard "hello, world" program
```

Notará que el uso de un repositorio de Mercurial utiliza el comando `git clone` estándar. Esto se debe a que git-remote-hg está funcionando a un nivel bastante bajo, utilizando un mecanismo similar a cómo se implementa el protocolo HTTP / S de Git (auxiliares remotos). Dado que Git y Mercurial están diseñados para que cada cliente tenga una copia completa del historial del repositorio, este comando hace un clon completo, incluyendo todo el historial del proyecto, y lo hace con bastante rapidez.

El comando de registro muestra dos confirmaciones, la última de las cuales es señalada por un montón de refs. Resulta que algunos de estos no están realmente allí. Echemos un vistazo a lo que realmente está en el directorio `.git`:

```
$ tree .git/refs
.git/refs
├── heads
│   └── master
├── hg
│   └── origin
│       ├── bookmarks
│       │   └── master
│       └── branches
│           └── default
├── notes
│   └── hg
├── remotes
│   └── origin
│       └── HEAD
└── tags

9 directories, 5 files
```

Git-remote-hg está tratando de hacer las cosas más idiomáticamente Git-esque, pero bajo el capó es la gestión de la cartografía conceptual entre dos sistemas ligeramente diferentes. El directorio `refs/hg` es donde se almacenan las referencias remotas reales. Por ejemplo, el `refs/hg/origen/branches/default` es un archivo ref de Git que contiene el SHA-1 que comienza con "ac7955c", que es el commit que señala `master`. Así que el directorio `refs/hg` es como un`refs/remotes/origen' falso, pero tiene la distinción añadida entre marcadores y ramas.

El archivo `notes/hg` es el punto de partida de cómo git-remote-hg asigna los hashes de commit de Git a los identificadores de cambios de Mercurial. Vamos a explorar un poco:

```
$ cat notes/hg
d4c10386...

$ git cat-file -p d4c10386...
tree 1781c96...
author remote-hg <> 1408066400 -0800
committer remote-hg <> 1408066400 -0800

Notes for master

$ git ls-tree 1781c96...
100644 blob ac9117f...  65bb417...
100644 blob 485e178...  ac7955c...

$ git cat-file -p ac9117f
0a04b987be5ae354b710cefeba0e2d9de7ad41a9
```

Así que `refs/notes/hg` apunta a un árbol, que en la base de datos de objetos Git es una lista de otros objetos con nombres. `Git ls-tree` genera el modo, el tipo, el hash de objeto y el nombre de archivo

de elementos dentro de un árbol. Una vez que excavamos hacia abajo a uno de los elementos del árbol, encontramos que en su interior hay un blob llamado “ac9117f” (el hash SHA-1 del commit apuntado por `master`), con contenidos “0a04b98” Que es el identificador del conjunto de cambios Mercurial en la punta de la rama `default`).

La buena noticia es que en general no tenemos que preocuparnos por todo esto. El flujo de trabajo típico no será muy diferente de trabajar con un control remoto de Git.

Hay una cosa más a la que debemos atender antes de continuar: ignora. Mercurial y Git usan un mecanismo muy similar para esto, pero es probable que no quieras realmente comprometer un archivo `.gitignore` en un repositorio de Mercurial. Afortunadamente, Git tiene una forma de ignorar los archivos que son locales a un repositorio en disco, y el formato Mercurial es compatible con Git, por lo que sólo tienes que copiarlo:

```
$ cp .hgignore .git/info/exclude
```

El archivo `.git / info / exclude 'actúa como un` .gitignore`, pero no está incluido en commits.

Flujo de Trabajo

Supongamos que hemos hecho algunos trabajos e hicimos algunos commit en la rama `master` y estamos listos para enviarlo al repositorio remoto. A continuación, le mostramos nuestro repositorio:

```
$ git log --oneline --graph --decorate
* ba04a2a (HEAD, master) Update makefile
* d25d16f Goodbye
* ac7955c (origin/master, origin/branches/default, origin/HEAD,
refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master) Create a makefile
* 65bb417 Create a standard "hello, world" program
```

Nuestra rama `master` está a dos compromisos por delante de `origin/master`, pero estos dos commits sólo existen en nuestra máquina local. Veamos si alguien más ha estado haciendo un trabajo importante al mismo tiempo:

```
$ git fetch
From hg::/tmp/hello
   ac7955c..df85e87  master     -> origin/master
   ac7955c..df85e87  branches/default -> origin/branches/default
$ git log --oneline --graph --decorate --all
* 7b07969 (refs/notes/hg) Notes for default
* d4c1038 Notes for master
* df85e87 (origin/master, origin/branches/default, origin/HEAD,
refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master) Add some
documentation
| * ba04a2a (HEAD, master) Update makefile
| * d25d16f Goodbye
|/
* ac7955c Create a makefile
* 65bb417 Create a standard "hello, world" program
```

Puesto que utilizamos el indicador `--all`, vemos las` notes '' refs que son utilizadas internamente por git-remote-hg, pero podemos ignorarlas. El resto es lo que esperábamos; `Origen / master ha avanzado por una comisión, y nuestra historia ha divergido ahora. A diferencia de los otros sistemas con los que trabajamos en este capítulo, Mercurial es capaz de manejar fusiones, por lo que no vamos a hacer nada extravagante.

```
$ git merge origin/master
Auto-merging hello.c
Merge made by the 'recursive' strategy.
 hello.c | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git log --oneline --graph --decorate
*   0c64627 (HEAD, master) Merge remote-tracking branch 'origin/master'
|\
| * df85e87 (origin/master, origin/branches/default, origin/HEAD,
refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master) Add some
documentation
* | ba04a2a Update makefile
* | d25d16f Goodbye
|/
* ac7955c Create a makefile
* 65bb417 Create a standard "hello, world" program
```

Perfecto. Hacemos las pruebas y todo pasa, así que estamos listos para compartir nuestro trabajo con el resto del equipo:

```
$ git push
To hg::/tmp/hello
   df85e87..0c64627  master -> master
```

¡Eso es! Si echa un vistazo al repositorio de Mercurial, verá que esto hizo lo que esperábamos:

```
$ hg log -G --style compact
o    5[tip]:4,2   dc8fa4f932b8   2014-08-14 19:33 -0700   ben
|\     Merge remote-tracking branch 'origin/master'
| |
| o  4   64f27bcefc35   2014-08-14 19:27 -0700   ben
| |    Update makefile
| |
| o  3:1   4256fc29598f   2014-08-14 19:27 -0700   ben
| |    Goodbye
| |
@ |  2   7db0b4848b3c   2014-08-14 19:30 -0700   ben
|/     Add some documentation
|
o  1   82e55d328c8c   2005-08-26 01:21 -0700   mpm
|    Create a makefile
|
o  0   0a04b987be5a   2005-08-26 01:20 -0700   mpm
     Create a standard "hello, world" program
```

El conjunto de cambios numerado *2* fue hecho por Mercurial, y los conjuntos de cambios numerados *3* y *4* fueron hechos por git-remote-hg, al empujar los commit hechos con Git.

Branches(Ramas) y Bookmarks(Marcadores)

Git tiene sólo un tipo de rama: una referencia que se mueve cuando se hacen los compromisos. En Mercurial, este tipo de referencia se llama marcador, y se comporta de la misma manera que una rama de Git.

El concepto de Mercurial de una "rama" es más pesado. La rama en la que se realiza un conjunto de cambios se registra con el conjunto de cambios, lo que significa que siempre estará en el historial del repositorio. He aquí un ejemplo de un commit que se hizo en la rama `develop`:

```
$ hg log -l 1
changeset:   6:8f65e5e02793
branch:      develop
tag:         tip
user:        Ben Straub <ben@straub.cc>
date:        Thu Aug 14 20:06:38 2014 -0700
summary:     More documentation
```

Observe la línea que comienza con “Branch”. Git no puede realmente replicar esto (y no necesita, ambos tipos de rama puede representarse como una referencia Git), pero git-remote-hg necesita entender la diferencia, porque Mercurial se preocupa.

Crear marcadores de Mercurial es tan fácil como crear ramas de Git. En el lado Git:

```
$ git checkout -b featureA
Switched to a new branch 'featureA'
$ git push origin featureA
To hg::/tmp/hello
 * [new branch]      featureA -> featureA
```

Eso es todo al respecto. En el lado mercurial, se ve así:

```
$ hg bookmarks
   featureA                  5:bd5ac26f11f9
$ hg log --style compact -G
@  6[tip]   8f65e5e02793   2014-08-14 20:06 -0700   ben
|    More documentation
|
o    5[featureA]:4,2   bd5ac26f11f9   2014-08-14 20:02 -0700   ben
|\     Merge remote-tracking branch 'origin/master'
| |
| o  4   0434aaa6b91f   2014-08-14 20:01 -0700   ben
| |    update makefile
| |
| o  3:1   318914536c86   2014-08-14 20:00 -0700   ben
| |    goodbye
| |
o |  2   f098c7f45c4f   2014-08-14 20:01 -0700   ben
|/     Add some documentation
|
o  1   82e55d328c8c   2005-08-26 01:21 -0700   mpm
|    Create a makefile
|
o  0   0a04b987be5a   2005-08-26 01:20 -0700   mpm
     Create a standard "hello, world" program
```

Tenga en cuenta la nueva etiqueta `[featureA]` en la revisión 5. Éstos actúan exactamente como las ramas de Git en el lado de Git, con una excepción: usted no puede suprimir un marcador del lado de Git (ésta es una limitación de ayudantes remotos).

Puedes trabajar con una rama “heavyweight” de Mercurial si: introduces ramas en los espacios para `branches` así:

```
$ git checkout -b branches/permanent
Switched to a new branch 'branches/permanent'
$ vi Makefile
$ git commit -am 'A permanent change'
$ git push origin branches/permanent
To hg::/tmp/hello
 * [new branch]      branches/permanent -> branches/permanent
```

Esto es lo que aparece en el lado de Mercurial:

```
$ hg branches
permanent                    7:a4529d07aad4
develop                      6:8f65e5e02793
default                      5:bd5ac26f11f9 (inactive)
$ hg log -G
o  changeset:   7:a4529d07aad4
|  branch:      permanent
|  tag:         tip
|  parent:      5:bd5ac26f11f9
|  user:        Ben Straub <ben@straub.cc>
|  date:        Thu Aug 14 20:21:09 2014 -0700
|  summary:     A permanent change
|
| @  changeset:   6:8f65e5e02793
|/   branch:      develop
|    user:        Ben Straub <ben@straub.cc>
|    date:        Thu Aug 14 20:06:38 2014 -0700
|    summary:     More documentation
|
o    changeset:   5:bd5ac26f11f9
|\   bookmark:    featureA
| |  parent:      4:0434aaa6b91f
| |  parent:      2:f098c7f45c4f
| |  user:        Ben Straub <ben@straub.cc>
| |  date:        Thu Aug 14 20:02:21 2014 -0700
| |  summary:     Merge remote-tracking branch 'origin/master'
[...]
```

El nombre de la rama “permanent” se registró en el conjunto de cambios marcados con *7*.

Desde el lado de Git, el trabajo con cualquiera de estos estilos de rama es el mismo: sólo checkout, commit, fetch, merge, pull y push como lo haría normalmente. Una cosa que usted debe saber es que Mercurial no apoya la historia de la reescritura, agregando solamente a ella. Esto es lo que nuestro repositorio de Mercurial parece después de un rebase interactivo y un force-push:

```
$ hg log --style compact -G
o  10[tip]   99611176cbc9   2014-08-14 20:21 -0700   ben
|    A permanent change
|
o  9   f23e12f939c3   2014-08-14 20:01 -0700   ben
|    Add some documentation
|
o  8:1   c16971d33922   2014-08-14 20:00 -0700   ben
|    goodbye
|
| o  7:5   a4529d07aad4   2014-08-14 20:21 -0700   ben
| |    A permanent change
| |
| | @  6   8f65e5e02793   2014-08-14 20:06 -0700   ben
| |/     More documentation
| |
| o    5[featureA]:4,2   bd5ac26f11f9   2014-08-14 20:02 -0700   ben
| |\     Merge remote-tracking branch 'origin/master'
| | |
| | o  4   0434aaa6b91f   2014-08-14 20:01 -0700   ben
| | |    update makefile
| | |
+---o  3:1   318914536c86   2014-08-14 20:00 -0700   ben
| |      goodbye
| |
| o  2   f098c7f45c4f   2014-08-14 20:01 -0700   ben
|/     Add some documentation
|
o  1   82e55d328c8c   2005-08-26 01:21 -0700   mpm
|    Create a makefile
|
o  0   0a04b987be5a   2005-08-26 01:20 -0700   mpm
     Create a standard "hello, world" program
```

CHangesets *8*, *9* y *10* han sido creados y pertenecen a la rama `permanent`, pero los viejos changesets siguen ahí. Esto puede ser **muy** confuso para sus compañeros de equipo que están usando Mercurial, así que trate de evitarlo.

Resumen de Mercurial

Git y Mercurial son bastante similares que trabajar a través de la frontera es bastante indoloro. Si evita cambiar el historial que ha dejado su máquina (como se recomienda generalmente), puede que ni siquiera sepa que el otro extremo es Mercurial.

Git y Perforce

Perforce es un sistema de control de versiones muy popular en entornos corporativos. Ha existido desde 1995, lo que lo convierte en el sistema más antiguo cubierto en este capítulo. Como tal, está diseñado con las limitaciones de su día; supone que siempre está conectado a un solo servidor

central y solo se conserva una versión en el disco local. Para estar seguro, sus características y limitaciones son adecuadas para varios problemas específicos, pero hay muchos proyectos que usan Perforce donde Git realmente funcionaría mejor.

Hay dos opciones si desea mezclar el uso de Perforce y Git. La primera que veremos es el puente “Git Fusion” de los creadores de Perforce, que le permite exponer los subárboles de su depósito de Perforce como repositorios de lectura y escritura de Git. La segunda es git-p4, un puente del lado del cliente que le permite usar Git como un cliente Perforce, sin requerir ninguna reconfiguración del servidor Perforce.

Git Fusion

Perforce proporciona un producto llamado Git Fusion (disponible en http://www.perforce.com/git-fusion), que sincroniza un servidor Perforce con repositorios Git en el lado del servidor.

Configurando

Para nuestros ejemplos, utilizaremos el método de instalación más fácil para Git Fusion, que está descargando una máquina virtual que ejecuta Perforce Daemon y Git Fusion. Puede obtener la imagen de la máquina virtual desde http://www.perforce.com/downloads/Perforce/20-User, y una vez que haya finalizado la descarga, impórtela en su software de virtualización favorito (utilizaremos VirtualBox).

Al iniciar la máquina por primera vez, le solicita que personalice la contraseña para tres usuarios de Linux (`root`,` perforce` y `git`), y proporcione un nombre de instancia, que se puede usar para distinguir esta instalación de otras en el misma red. Cuando todo haya terminado, verás esto:

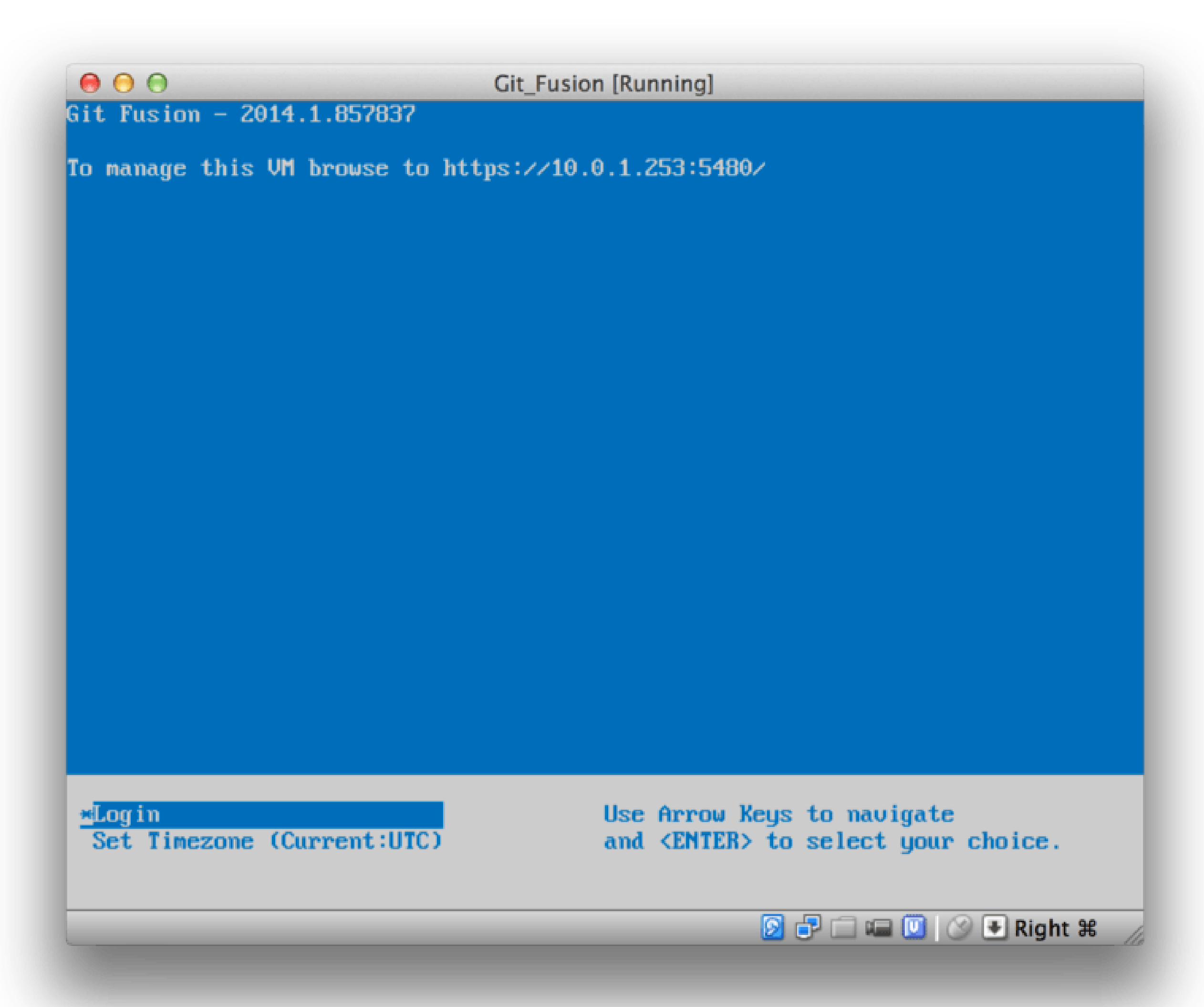


Figure 146. The Git Fusion virtual machine boot screen.

Debe tomar nota de la dirección IP que se muestra aquí, la usaremos más adelante. A continuación, crearemos un usuario de Perforce. Seleccione la opción “Iniciar sesión” en la parte inferior y presione enter (o SSH en la máquina) e inicie sesión como `root`. Luego use estos comandos para crear un usuario:

```
$ p4 -p localhost:1666 -u super user -f john
$ p4 -p localhost:1666 -u john passwd
$ exit
```

El primero abrirá un editor VI para personalizar al usuario, pero puede aceptar los valores predeterminados escribiendo `:wq` y pulsando enter. El segundo le pedirá que ingrese una contraseña dos veces. Eso es todo lo que tenemos que hacer con un intérprete de comandos de shell, así que salga de la sesión.

Lo siguiente que tendrá que hacer para seguirlo es decirle a Git que no verifique los certificados SSL. La imagen de Git Fusion viene con un certificado, pero es para un dominio que no coincidirá con la dirección IP de su máquina virtual, por lo que Git rechazará la conexión HTTPS. Si va a ser una instalación permanente, consulte el manual de Perforce de Git Fusion para instalar un

certificado diferente; para nuestros propósitos de ejemplo, esto será suficiente:

```
$ export GIT_SSL_NO_VERIFY=true
```

Ahora podemos probar que todo está funcionando.

```
$ git clone https://10.0.1.254/Talkhouse
Cloning into 'Talkhouse'...
Username for 'https://10.0.1.254': john
Password for 'https://john@10.0.1.254':
remote: Counting objects: 630, done.
remote: Compressing objects: 100% (581/581), done.
remote: Total 630 (delta 172), reused 0 (delta 0)
Receiving objects: 100% (630/630), 1.22 MiB | 0 bytes/s, done.
Resolving deltas: 100% (172/172), done.
Checking connectivity... done.
```

La imagen de la máquina virtual viene equipada con un proyecto de muestra que puede clonar. Aquí estamos clonando a través de HTTPS, con el usuario `john` que creamos anteriormente; Git solicita credenciales para esta conexión, pero la caché de credenciales nos permitirá omitir este paso para cualquier solicitud posterior.

Configuración de Fusion

Una vez que haya instalado Git Fusion, querrá modificar la configuración.Esto es bastante fácil de hacer usando su cliente Perforce favorito; simplemente asigne el directorio `//.git-fusion` en el servidor Perforce en su espacio de trabajo.La estructura del archivo se ve así:

```
$ tree
.
├── objects
│   ├── repos
│   │   └── [...]
│   └── trees
│       └── [...]
│
├── p4gf_config
├── repos
│   └── Talkhouse
│       └── p4gf_config
└── users
    └── p4gf_usermap

498 directories, 287 files
```

El directorio `objects` es usado internamente por Git Fusion para asignar objetos Perforce a Git y viceversa, no tendrá que meterse con nada allí. Hay un archivo global `p4gf_config` en este

directorio, así como uno para cada repositorio – estos son los archivos de configuración que determinan cómo se comporta Git Fusion. Echemos un vistazo al archivo en la raíz:

```
[repo-creation]
charset = utf8

[git-to-perforce]
change-owner = author
enable-git-branch-creation = yes
enable-swarm-reviews = yes
enable-git-merge-commits = yes
enable-git-submodules = yes
preflight-commit = none
ignore-author-permissions = no
read-permission-check = none
git-merge-avoidance-after-change-num = 12107

[perforce-to-git]
http-url = none
ssh-url = none

[@features]
imports = False
chunked-push = False
matrix2 = False
parallel-push = False

[authentication]
email-case-sensitivity = no
```

No entraremos en el significado de estos indicadores aquí, pero tenga en cuenta que esto es solo un archivo de texto con formato INI, muy parecido al que Git usa para la configuración. Este archivo especifica las opciones globales, que luego pueden ser reemplazadas por archivos de configuración específicos del repositorio, como `repos/Talkhouse/p4gf_config`. Si abre este archivo, verá una sección `[@repo]` con algunas configuraciones que son diferentes de los valores predeterminados globales. También verá secciones que se ven así:

```
[Talkhouse-master]
git-branch-name = master
view = //depot/Talkhouse/main-dev/... ...
```

Este es un mapeo entre una rama Perforce y una rama Git. La sección se puede nombrar como prefiera, siempre que el nombre sea único. `git-branch-name` le permite convertir una ruta de depósito que sería engorrosa bajo Git a un nombre más amigable. La configuración `view` controla cómo se asocian los archivos de Perforce en el repositorio de Git, usando la sintaxis de mapeo de vista estándar. Se puede especificar más de un mapeo, como en este ejemplo:

```
[multi-project-mapping]
git-branch-name = master
view = //depot/project1/main/... project1/...
       //depot/project2/mainline/... project2/...
```

De esta manera, si la asignación normal del espacio de trabajo incluye cambios en la estructura de los directorios, puede replicar eso con un repositorio Git.

El último archivo que discutiremos es `users/p4gf_usermap`, que mapea a los usuarios de Perforce a los usuarios de Git, y que quizás ni siquiera necesites. Al convertir de un conjunto de cambios de Perforce a una commit de Git, el comportamiento predeterminado de Git Fusion es buscar al usuario de Perforce y usar la dirección de correo electrónico y el nombre completo almacenados allí para el campo autor/committer en Git. Al realizar la conversión de otra manera, el valor predeterminado es buscar al usuario de Perforce con la dirección de correo electrónico almacenada en el campo de autoría del commit de Git y enviar el conjunto de cambios como ese usuario (con la aplicación de permisos). En la mayoría de los casos, este comportamiento funcionará bien, pero considere el siguiente archivo de mapeo:

```
john john@example.com "John Doe"
john johnny@appleseed.net "John Doe"
bob employeeX@example.com "Anon X. Mouse"
joe employeeY@example.com "Anon Y. Mouse"
```

Cada línea tiene el formato <usuario> <correo electrónico> "<nombre completo>" y crea una sola asignación de usuario.Las dos primeras líneas asignan dos direcciones de correo electrónico distintas a la misma cuenta de usuario de Perforce. Esto es útil si ha creado commits de Git en varias direcciones de correo electrónico diferentes (o cambia direcciones de correo electrónico), pero quiere que se mapeen al mismo usuario de Perforce. Al crear una commit de Git a partir de un conjunto de cambios de Perforce, la primera línea que coincide con el usuario de Perforce se utiliza para la información de autoría de Git.

Las últimas dos líneas ocultan los nombres reales y las direcciones de correo electrónico de Bob y Joe de las commits de Git que se crean. Esto es bueno si desea abrir un proyecto interno de fuente abierta, pero no desea publicar su directorio de empleados en todo el mundo. Tenga en cuenta que las direcciones de correo electrónico y los nombres completos deben ser únicos, a menos que desee que todos los commit de Git se atribuyan a un único autor ficticio.

Flujo de trabajo

Perforce de Git Fusion es un puente de dos vías entre Perforce y el control de versiones de Git. Echemos un vistazo a cómo se siente trabajar desde el lado de Git. Asumiremos que hemos mapeado en el proyecto “Jam” usando un archivo de configuración como se muestra arriba, el cual podemos clonar así:

```
$ git clone https://10.0.1.254/Jam
Cloning into 'Jam'...
Username for 'https://10.0.1.254': john
Password for 'https://ben@10.0.1.254':
remote: Counting objects: 2070, done.
remote: Compressing objects: 100% (1704/1704), done.
Receiving objects: 100% (2070/2070), 1.21 MiB | 0 bytes/s, done.
remote: Total 2070 (delta 1242), reused 0 (delta 0)
Resolving deltas: 100% (1242/1242), done.
Checking connectivity... done.
$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/rel2.1
$ git log --oneline --decorate --graph --all
* 0a38c33 (origin/rel2.1) Create Jam 2.1 release branch.
| * d254865 (HEAD, origin/master, origin/HEAD, master) Upgrade to latest metrowerks on
Beos -- the Intel one.
| * bd2f54a Put in fix for jam's NT handle leak.
| * c0f29e7 Fix URL in a jam doc
| * cc644ac Radstone's lynx port.
[...]
```

La primera vez que haces esto, puede tomar algún tiempo. Lo que sucede es que Git Fusion está convirtiendo todos los conjuntos de cambios aplicables en el historial de Perforce en commits de Git. Esto ocurre localmente en el servidor, por lo que es relativamente rápido, pero si tienes un montón de historia, aún puede tomar algo de tiempo. Las recuperaciones posteriores realizan conversiones incrementales, por lo que se parecerá más a la velocidad nativa de Git.

Como puede ver, nuestro repositorio se ve exactamente como cualquier otro repositorio de Git con el que pueda trabajar. Hay tres ramas, y Git ha creado una rama `master` local que rastrea `origin/master`. Hagamos un poco de trabajo y creemos un par de nuevos commits:

```
# ...
$ git log --oneline --decorate --graph --all
* cfd46ab (HEAD, master) Add documentation for new feature
* a730d77 Whitespace
* d254865 (origin/master, origin/HEAD) Upgrade to latest metrowerks on Beos -- the
Intel one.
* bd2f54a Put in fix for jam's NT handle leak.
[...]
```

Tenemos dos nuevos commits. Ahora revisemos si alguien más ha estado trabajando:

```
$ git fetch
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://10.0.1.254/Jam
   d254865..6afeb15  master     -> origin/master
$ git log --oneline --decorate --graph --all
* 6afeb15 (origin/master, origin/HEAD) Update copyright
| * cfd46ab (HEAD, master) Add documentation for new feature
| * a730d77 Whitespace
|/
* d254865 Upgrade to latest metrowerks on Beos -- the Intel one.
* bd2f54a Put in fix for jam's NT handle leak.
[...]
```

¡Parece que alguien lo está! No lo sabrías desde esta vista, pero la commit `6afeb15` se creó realmente utilizando un cliente Perforce. Solo parece otra commit desde el punto de vista de Git, que es exactamente el punto. Veamos cómo el servidor Perforce trata con una commit de fusión:

```
$ git merge origin/master
Auto-merging README
Merge made by the 'recursive' strategy.
 README | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git push
Counting objects: 9, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (9/9), 917 bytes | 0 bytes/s, done.
Total 9 (delta 6), reused 0 (delta 0)
remote: Perforce: 100% (3/3) Loading commit tree into memory...
remote: Perforce: 100% (5/5) Finding child commits...
remote: Perforce: Running git fast-export...
remote: Perforce: 100% (3/3) Checking commits...
remote: Processing will continue even if connection is closed.
remote: Perforce: 100% (3/3) Copying changelists...
remote: Perforce: Submitting new Git commit objects to Perforce: 4
To https://10.0.1.254/Jam
   6afeb15..89cba2b  master -> master
```

Git cree que funcionó. Echemos un vistazo al historial del archivo `README` desde el punto de vista de Perforce, usando la función de gráfico de revisión de `p4v`:

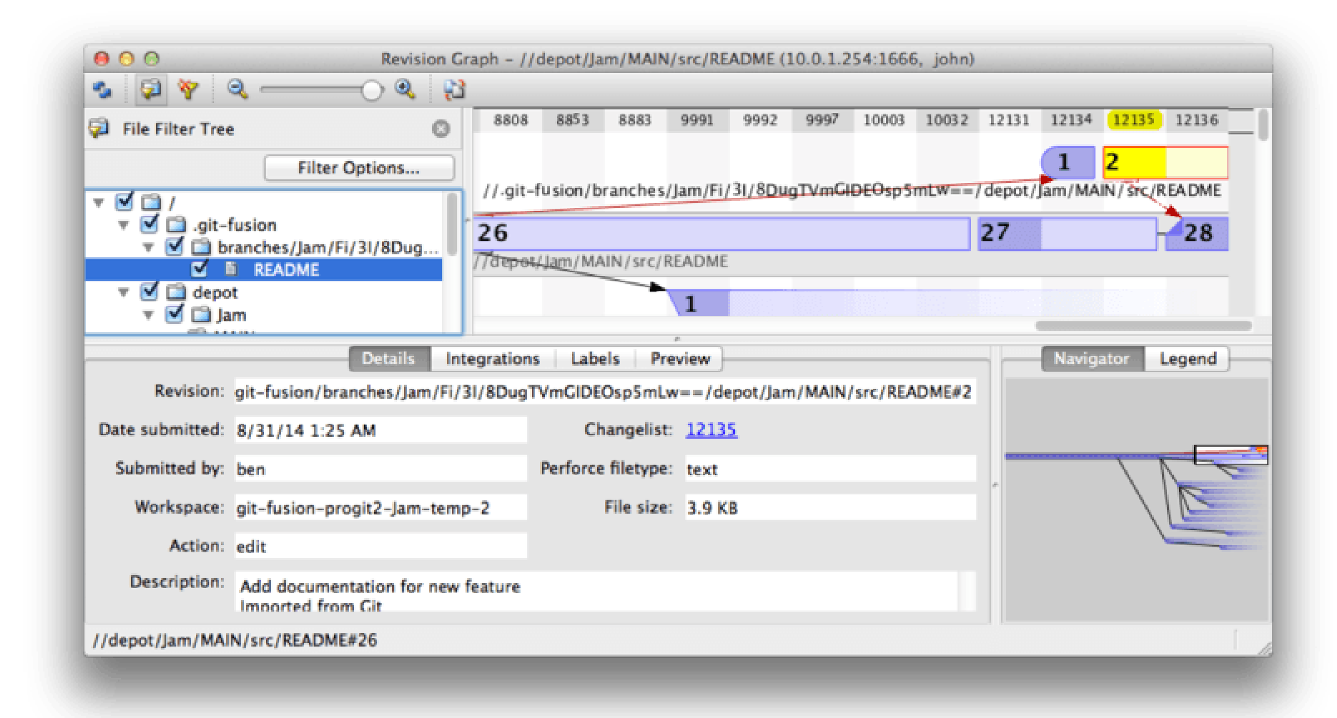


Figure 147. Perforce revision graph resulting from Git push.

Si nunca antes has visto esta vista, puede parecer confusa, pero muestra los mismos conceptos que un visor gráfico para el historial de Git. Estamos viendo el historial del archivo `README`, por lo que el árbol de directorios en la parte superior izquierda solo muestra ese archivo a medida que aparece en varias ramas. En la parte superior derecha, tenemos un gráfico visual de cómo se relacionan las diferentes revisiones del archivo, y la vista general de este gráfico se encuentra en la parte inferior derecha. El resto de la vista se da a la vista de detalles para la revisión seleccionada (`2` en este caso).

Una cosa para notar es que el gráfico se ve exactamente como el del historial de Git. Perforce no tenía una rama con nombre para almacenar las commits `1` y `2`, por lo que creó una rama “anónima” en el directorio `.git-fusion` para contenerla. Esto también ocurrirá para las ramas nombradas de Git que no se corresponden con una rama de Perforce con nombre (y luego puede asignarlas a una rama de Perforce usando el archivo de configuración).

La mayoría de esto sucede detrás de escena, pero el resultado final es que una persona en un equipo puede estar usando Git, otra puede estar usando Perforce, y ninguno de ellos conocerá la elección del otro.

Resumen de Git-Fusion

Si tiene (o puede obtener) acceso a su servidor Perforce, Git Fusion es una excelente manera de hacer que Git y Perforce hablen entre sí. Hay un poco de configuración involucrada, pero la curva de aprendizaje no es muy pronunciada. Esta es una de las pocas secciones en este capítulo donde las precauciones sobre el uso de la potencia total de Git no aparecerán. Eso no quiere decir que Perforce esté contento con todo lo que le arrojas – si tratas de reescribir la historia que ya ha sido empujada, Git Fusion la rechazará – pero Git Fusion trata muy fuertemente de sentirse nativo. Incluso puede usar submódulos de Git (aunque parecerán extraños a los usuarios de Perforce) y unir ramas (esto se registrará como una integración en el lado de Perforce).

Si no puede convencer al administrador de su servidor para configurar Git Fusion, todavía hay una

manera de utilizar estas herramientas juntas.

Git-p4

Git-p4 es un puente de dos vías entre Git y Perforce. Funciona completamente dentro de su repositorio Git, por lo que no necesitará ningún tipo de acceso al servidor Perforce (aparte de las credenciales de usuario, por supuesto). Git-p4 no es tan flexible ni completa una solución como Git Fusion, pero le permite hacer la mayor parte de lo que le gustaría hacer sin ser invasivo en el entorno del servidor.

NOTE	Necesitará la herramienta `p4` en algún lugar de su` PATH` para trabajar con git-p4. Al momento de escribir esto, está disponible gratuitamente en http://www.perforce.com/downloads/Perforce/20-User.

Configurando

Por ejemplo, ejecutaremos el servidor Perforce desde Git Fusion OVA como se muestra arriba, pero omitiremos el servidor de Git Fusion y pasaremos directamente al control de versión de Perforce.

Para utilizar el cliente de línea de comandos `p4` (del cual depende git-p4), deberá establecer un par de variables de entorno:

```
$ export P4PORT=10.0.1.254:1666
$ export P4USER=john
```

Empezando

Al igual que con cualquier cosa en Git, el primer comando es clonar:

```
$ git p4 clone //depot/www/live www-shallow
Importing from //depot/www/live into www-shallow
Initialized empty Git repository in /private/tmp/www-shallow/.git/
Doing initial import of //depot/www/live/ from revision #head into
refs/remotes/p4/master
```

Esto crea lo que en términos de Git es un clon “superficial”; solo la última versión de Perforce se importa a Git; recuerde, Perforce no está diseñado para dar cada revisión a cada usuario. Este es suficiente para usar Git como cliente de Perforce, pero para otros fines no es suficiente.

Una vez que está terminado, tenemos un repositorio de Git completamente funcional:

```
$ cd myproject
$ git log --oneline --all --graph --decorate
* 70eaf78 (HEAD, p4/master, p4/HEAD, master) Initial import of //depot/www/live/ from
the state at revision #head
```

Tenga en cuenta que hay un control remoto “p4” para el servidor de Perforce, pero todo lo demás

parece un clon estándar. En realidad, eso es un poco engañoso; no hay realmente un control remoto allí.

```
$ git remote -v
```

No hay controles remotos en este repositorio en lo absoluto. Git-p4 ha creado algunas refs para representar el estado del servidor, y se ven como refs remotas para `git log`, pero no son administradas por Git, y no puedes presionarlas.

Flujo de trabajo

De acuerdo, hagamos un poco de trabajo. Supongamos que ha hecho algún progreso en una característica muy importante y está listo para mostrársela al resto de su equipo.

```
$ git log --oneline --all --graph --decorate
* 018467c (HEAD, master) Change page title
* c0fb617 Update link
* 70eaf78 (p4/master, p4/HEAD) Initial import of //depot/www/live/ from the state at
revision #head
```

Hemos realizado dos nuevas commits que estamos listos para enviar al servidor de Perforce. Comprobemos si alguien más estaba trabajando hoy:

```
$ git p4 sync
git p4 sync
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
Import destination: refs/remotes/p4/master
Importing revision 12142 (100%)
$ git log --oneline --all --graph --decorate
* 75cd059 (p4/master, p4/HEAD) Update copyright
| * 018467c (HEAD, master) Change page title
| * c0fb617 Update link
|/
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

Parece que si, y `master` y `p4/master` han divergido. El sistema de ramificación de Perforce es *nada* similar al de Git, por lo que enviar commits de fusión no tiene ningún sentido. Git-p4 recomienda que rebases tus commits, e incluso viene con un atajo para hacerlo:

```
$ git p4 rebase
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
No changes to import!
Rebasing the current branch onto remotes/p4/master
First, rewinding head to replay your work on top of it...
Applying: Update link
Applying: Change page title
 index.html | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Probablemente pueda decir desde la salida, pero `git p4 rebase` es un atajo para` git p4 sync` seguido de `git rebase p4/master`. Es un poco más inteligente que eso, especialmente cuando se trabaja con múltiples ramas, pero esta es una buena aproximación.

Ahora nuestra historia es lineal nuevamente y estamos listos para contribuir nuestros cambios de vuelta en Perforce. El comando `git p4 submit` intentará crear una nueva revisión de Perforce para cada commit de Git entre `p4/master` y `master`. Al ejecutarlo, nos deja en nuestro editor favorito, y los contenidos del archivo se ven algo así:

```
# A Perforce Change Specification.
#
#  Change:      The change number. 'new' on a new changelist.
#  Date:        The date this specification was last modified.
#  Client:      The client on which the changelist was created.  Read-only.
#  User:        The user who created the changelist.
#  Status:      Either 'pending' or 'submitted'. Read-only.
#  Type:        Either 'public' or 'restricted'. Default is 'public'.
#  Description: Comments about the changelist.  Required.
#  Jobs:        What opened jobs are to be closed by this changelist.
#               You may delete jobs from this list.  (New changelists only.)
#  Files:       What opened files from the default changelist are to be added
#               to this changelist.  You may delete files from this list.
#               (New changelists only.)

Change:  new

Client:  john_bens-mbp_8487

User: john

Status:  new

Description:
   Update link

Files:
   //depot/www/live/index.html   # edit


######## git author ben@straub.cc does not match your p4 account.
######## Use option --preserve-user to modify authorship.
######## Variable git-p4.skipUserNameCheck hides this message.
######## everything below this line is just the diff #######
--- //depot/www/live/index.html  2014-08-31 18:26:05.000000000 0000
+++ /Users/ben/john_bens-mbp_8487/john_bens-mbp_8487/depot/www/live/index.html   2014-
08-31 18:26:05.000000000 0000
@@ -60,7 +60,7 @@
 </td>
 <td valign=top>
 Source and documentation for
-<a href="http://www.perforce.com/jam/jam.html">
+<a href="jam.html">
 Jam/MR</a>,
 a software build tool.
 </td>
```

Esto es principalmente el mismo contenido que verías al ejecutar `p4 submit`, excepto las cosas al final que git-p4 ha incluido amablemente. Git-p4 intenta honrar su configuración de Git y Perforce

individualmente cuando tiene que proporcionar un nombre para un commit o un conjunto de cambios, pero en algunos casos usted quiere anularla. Por ejemplo, si el commit de Git que está importando fue escrito por un colaborador que no tiene una cuenta de usuario de Perforce, es posible que aún quiera que el conjunto de cambios resultante tenga el aspecto de que lo escribió (y no usted).

Git-p4 ha importado amablemente el mensaje de la confirmación de Git como el contenido de este conjunto de cambios de Perforce, por lo que todo lo que tenemos que hacer es guardar y salir, dos veces (una para cada confirmación). El resultado del shell resultante será algo como esto:

```
$ git p4 submit
Perforce checkout for depot path //depot/www/live/ located at /Users/ben/john_bens-
mbp_8487/john_bens-mbp_8487/depot/www/live/
Synchronizing p4 checkout...
... - file(s) up-to-date.
Applying dbac45b Update link
//depot/www/live/index.html#4 - opened for edit
Change 12143 created with 1 open file(s).
Submitting change 12143.
Locking 1 files ...
edit //depot/www/live/index.html#5
Change 12143 submitted.
Applying 905ec6a Change page title
//depot/www/live/index.html#5 - opened for edit
Change 12144 created with 1 open file(s).
Submitting change 12144.
Locking 1 files ...
edit //depot/www/live/index.html#6
Change 12144 submitted.
All commits applied!
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
Import destination: refs/remotes/p4/master
Importing revision 12144 (100%)
Rebasing the current branch onto remotes/p4/master
First, rewinding head to replay your work on top of it...
$ git log --oneline --all --graph --decorate
* 775a46f (HEAD, p4/master, p4/HEAD, master) Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

El resultado es como si hubiéramos hecho un `git push`, que es la analogía más cercana a lo que realmente sucedió.

Tenga en cuenta que durante este proceso, cada commit de Git se convierte en un conjunto de cambios de Perforce; si desea aplastarlos en un único conjunto de cambios, puede hacerlo con una rebase interactiva antes de ejecutar `git p4 submit`. También tenga en cuenta que los hashes SHA-1 de todas las commits que se enviaron como conjuntos de cambios han cambiado; esto es porque git-

p4 agrega una línea al final de cada confirmación que convierte:

```
$ git log -1
commit 775a46f630d8b46535fc9983cf3ebe6b9aa53145
Author: John Doe <john@example.com>
Date:   Sun Aug 31 10:31:44 2014 -0800

    Change page title

    [git-p4: depot-paths = "//depot/www/live/": change = 12144]
```

¿Qué sucede si intentas enviar una commit de fusión? Hagamos un intento. Esta es la situación en la que nos hemos metido:

```
$ git log --oneline --all --graph --decorate
* 3be6fd8 (HEAD, master) Correct email address
*   1dcbf21 Merge remote-tracking branch 'p4/master'
|\
| * c4689fc (p4/master, p4/HEAD) Grammar fix
* | cbacd0a Table borders: yes please
* | b4959b6 Trademark
|/
* 775a46f Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

La historia de Git y Perforce divergen después de 775a46f. El lado de Git tiene dos commits, luego un commit de fusión con la cabeza de Perforce, y luego otro commit. Vamos a tratar de enviar estos sobre un único conjunto de cambios en el lado de Perforce. Veamos qué pasaría si intentáramos enviar ahora:

```
$ git p4 submit -n
Perforce checkout for depot path //depot/www/live/ located at /Users/ben/john_bens-
mbp_8487/john_bens-mbp_8487/depot/www/live/
Would synchronize p4 checkout in /Users/ben/john_bens-mbp_8487/john_bens-
mbp_8487/depot/www/live/
Would apply
  b4959b6 Trademark
  cbacd0a Table borders: yes please
  3be6fd8 Correct email address
```

La bandera `-n` es la abreviatura de `--dry-run`, que intenta informar qué pasaría si el comando submit se ejecutara de manera real. En este caso, parece que estaríamos creando tres conjuntos de cambios de Perforce, que corresponden a las tres confirmaciones que no se fusionan y que todavía no existen en el servidor Perforce. Eso suena exactamente como lo que queremos, veamos cómo resulta:

```
$ git p4 submit
[…]
$ git log --oneline --all --graph --decorate
* dadbd89 (HEAD, p4/master, p4/HEAD, master) Correct email address
* 1b79a80 Table borders: yes please
* 0097235 Trademark
* c4689fc Grammar fix
* 775a46f Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

Nuestra historia se volvió lineal, como si hubiéramos vuelto a basar antes de enviar (que de hecho es exactamente lo que sucedió). Esto significa que puede ser libre para crear, trabajar, deshacer y fusionar ramas en el lado de Git sin miedo a que su historia se vuelva incompatible con Perforce. Si puede volver a establecerla, puede contribuir a un servidor de Perforce.

Ramificación

Si su proyecto Perforce tiene múltiples ramas, no está sin suerte; git-p4 puede manejar eso de una manera que lo haga sentir como Git. Digamos que su depósito Perforce se presenta así:

```
//depot
  └── project
      ├── main
      └── dev
```

Y digamos que tiene una rama `dev`, que tiene una especificación de vista que se ve así:

```
//depot/project/main/... //depot/project/dev/...
```

Git-p4 can automatically detect that situation and do the right thing:

```
$ git p4 clone --detect-branches //depot/project@all
Importing from //depot/project@all into project
Initialized empty Git repository in /private/tmp/project/.git/
Importing revision 20 (50%)
    Importing new branch project/dev

    Resuming with change 20
Importing revision 22 (100%)
Updated branches: main dev
$ cd project; git log --oneline --all --graph --decorate
* eae77ae (HEAD, p4/master, p4/HEAD, master) main
| * 10d55fb (p4/project/dev) dev
| * a43cfae Populate //depot/project/main/... //depot/project/dev/....
|/
* 2b83451 Project init
```

Tenga en cuenta el especificador “@all” en la ruta de depósito; eso le dice a git-p4 que clone no solo el último conjunto de cambios para ese subárbol, sino todos los conjuntos de cambios que alguna vez hayan tocado esas rutas. Esto está más cerca del concepto de clon de Git, pero si está trabajando en un proyecto con una larga historia, podría llevar un tiempo.

La bandera `--detect-branches` le dice a git-p4 que use las especificaciones de rama de Perforce para asignar las ramas a las refs de Git. Si estas asignaciones no están presentes en el servidor Perforce (que es una forma perfectamente válida de usar Perforce), puede indicar a git-p4 cuáles son las asignaciones de bifurcación y obtendrá el mismo resultado:

```
$ git init project
Initialized empty Git repository in /tmp/project/.git/
$ cd project
$ git config git-p4.branchList main:dev
$ git clone --detect-branches //depot/project@all .
```

Estableciendo la variable de configuración `git-p4.branchList` en `main:dev` le dice a git-p4 que “main” y “dev” son ambas ramas, y la segunda es hija de la primera.

Si ahora `git checkout -b dev p4/project/dev` y realizamos algunas commits, git-p4 es lo suficientemente inteligente para apuntar a la rama currecta cuando hacemos `git p4 submit`. Desafortunadamente, git-p4 no puede mezclar clones superficiales y ramas múltiples; si tienes un gran proyecto y quieres trabajar en más de una, tendrás que *git p4 clone* una vez por cada rama a la que quieras enviar.

Para crear o integrar ramas, deberá usar un cliente Perforce. Git-p4 solo puede sincronizar y enviar a las sucursales existentes, y solo puede hacerlo un conjunto de cambios lineal a la vez. Si combina dos ramas en Git e intenta enviar el nuevo conjunto de cambios, todo lo que se registrará será un conjunto de cambios de archivos; los metadatos sobre qué ramas están involucradas en la integración se perderán.

Resumen de Git y Perforce

Git-p4 hace posible usar un flujo de trabajo de Git con un servidor Perforce, y es bastante bueno en eso. Sin embargo, es importante recordar que Perforce está a cargo de la fuente y que solo está usando Git para trabajar localmente. Solo ten mucho cuidado al compartir commits de Git; si tiene un control remoto que utilizan otras personas, no envíe ninguna commit que aún no se haya enviado al servidor de Perforce.

Si desea mezclar libremente el uso de Perforce y Git como clientes para el control de código fuente, y puede convencer al administrador del servidor para que lo instale, Git Fusion hace que Git sea un cliente de control de versiones de primera clase para un servidor Perforce.

Git y TFS

Git se está volviendo popular entre los desarolladores de Windows, y si usted está escribiendo código en Windows, hay una gran chance de que esté usando Team Foundation Server (TFS), de Microsoft. TFS is a collaboration suite that includes defect and work-item tracking, process support for Scrum and others, code review, and version control. There's a bit of confusion ahead: **TFS** is the server, which supports controlling source code using both Git and their own custom VCS, which they've dubbed **TFVC** (Team Foundation Version Control). Git support is a somewhat new feature for TFS (shipping with the 2013 version), so all of the tools that predate that refer to the version-control portion as "TFS", even though they're mostly working with TFVC.

Git se está volviendo popular entre los desarrolladores de Windows, y si estás escribiendo códigos en Windows, hay muchas posibilidades de que estés usando Team Foundation Server (TFS) de Microsoft. TFS es un paquete de colaboración que incluye seguimiento de defectos y elementos de trabajo, soporte de procesos para Scrum y otros, revisión de código y control de versiones. Hay un poco de confusión por delante: * TFS * es el servidor, que admite controlar el código fuente utilizando tanto Git como su propio VCS personalizado, al que han denominado * TFVC * (Team Foundation Version Control). El soporte de Git es una característica algo nueva para TFS (envío con la versión de 2013), por lo que todas las herramientas anteriores a eso se refieren a la porción de control de versión como ‘` TFS '', aunque en su mayoría funcionan con TFVC.

Si se encuentra en un equipo que usa TFVC pero prefiere usar Git como su cliente de control de versiones, hay un proyecto para usted.

Cual herramienta

De hecho, hay dos: git-tf y git-tfs.

Git-tfs (encontrado en https://github.com/git-tfs/git-tfs []) es un proyecto .NET, y (al momento de escribir esto) solo se ejecuta en Windows. Para trabajar con repositorios Git, utiliza los enlaces .NET para libgit2, una implementación de Git orientada a la biblioteca que es altamente eficiente y permite mucha flexibilidad con las agallas de un repositorio Git. Libgit2 no es una implementación completa de Git, por lo que para cubrir la diferencia, git-tfs realmente llamará al cliente Git de la línea de comandos para algunas operaciones, por lo que no hay límites artificiales sobre lo que puede hacer con los repositorios Git. Su compatibilidad con las características de TFVC es muy madura, ya que utiliza los ensamblados de Visual Studio para operaciones con servidores. Esto

significa que necesitará acceso a esos ensamblados, lo que significa que necesita instalar una versión reciente de Visual Studio (cualquier edición desde la versión 2010, incluido Express desde la versión 2012) o el SDK de Visual Studio.

Git-tf (cuyo domicilio se encuentra en https://gittf.codeplex.com []) es un proyecto de Java y, como tal, se ejecuta en cualquier computadora con un entorno de tiempo de ejecución de Java. Interactúa con los repositorios de Git a través de JGit (una implementación JVM de Git), lo que significa que prácticamente no tiene limitaciones en términos de funciones de Git. Sin embargo, su soporte para TFVC es limitado en comparación con git-tfs; por ejemplo, no admite sucursales.

Entonces, cada herramienta tiene ventajas y desventajas, y hay muchas situaciones que favorecen a una sobre la otra. Cubriremos el uso básico de ambos en este libro.

NOTE Necesitará acceder a un repositorio basado en TFVC para seguir estas instrucciones. Estos no son tan abundantes en la naturaleza como los repositorios de Git o Subversion, por lo que puede necesitar crear uno propio. Codeplex (https://www.codeplex.com []) o Visual Studio Online (http://www.visualstudio.com []) son buenas opciones para esto.

Comenzando con: `git-tf`

Lo primero que haces, al igual que con cualquier proyecto de Git, es clonar. Esto es lo que parece con `git-tf`:

```
$ git tf clone https://tfs.codeplex.com:443/tfs/TFS13 $/myproject/Main project_git
```

El primer argumento es la URL de una colección TFVC, el segundo es de la forma `$ / project / branch`, y el tercero es la ruta al repositorio Git local que se va a crear (este último es opcional). Git-tf solo puede funcionar con una rama a la vez; si quieres hacer checkins en una rama diferente de TFVC, tendrás que hacer un nuevo clon desde esa rama.

Esto crea un repositorio de Git completamente funcional:

```
$ cd project_git
$ git log --all --oneline --decorate
512e75a (HEAD, tag: TFS_C35190, origin_tfs/tfs, master) Checkin message
```

Esto se denomina clon *shallow*, lo que significa que solo se ha descargado el último conjunto de cambios. TFVC no está diseñado para que cada cliente tenga una copia completa del historial, por lo que git-tf usa de manera predeterminada la última versión, que es mucho más rápida.

Si tiene algo de tiempo, probablemente valga la pena clonar todo el historial del proyecto, usando la opción `--deep`:

```
$ git tf clone https://tfs.codeplex.com:443/tfs/TFS13 $/myproject/Main \
  project_git --deep
Username: domain\user
Password:
Connecting to TFS...
Cloning $/myproject into /tmp/project_git: 100%, done.
Cloned 4 changesets. Cloned last changeset 35190 as d44b17a
$ cd project_git
$ git log --all --oneline --decorate
d44b17a (HEAD, tag: TFS_C35190, origin_tfs/tfs, master) Goodbye
126aa7b (tag: TFS_C35189)
8f77431 (tag: TFS_C35178) FIRST
0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
        Team Project Creation Wizard
```

Observe las etiquetas con nombres como `TFS_C35189`; esta es una característica que te ayuda a saber qué compromisos de Git están asociados con los conjuntos de cambios de TFVC. Esta es una buena forma de representarlo, ya que puede ver con un comando de registro simple cuál de sus confirmaciones está asociado con una instantánea que también existe en TFVC. No son necesarios (y, de hecho, puedes desactivarlos con `git config git-tf.tag false`) - git-tf conserva las asignaciones reales commit-changeset en el archivo` .git / git-tf`.

Comenzando: `git-tfs`

La clonación de Git-tfs se comporta de forma un poco diferente. Observe:

```
PS> git tfs clone --with-branches \
    https://username.visualstudio.com/DefaultCollection \
    $/project/Trunk project_git
Initialized empty Git repository in C:/Users/ben/project_git/.git/
C15 = b75da1aba1ffb359d00e85c52acb261e4586b0c9
C16 = c403405f4989d73a2c3c119e79021cb2104ce44a
Tfs branches found:
- $/tfvc-test/featureA
The name of the local branch will be : featureA
C17 = d202b53f67bde32171d5078968c644e562f1c439
C18 = 44cd729d8df868a8be20438fdeeefb961958b674
```

Observe el indicador `--with-branches`. Git-tfs es capaz de mapear ramas de TFVC a ramas de Git, y este indicador le dice que configure una rama local de Git para cada rama de TFVC. Esto es muy recomendable si alguna vez se ha bifurcado o se ha fusionado en TFS, pero no funcionará con un servidor anterior a TFS 2010; antes de esa versión, ‘` branches '' eran solo carpetas, por lo que git-tfs no puede decir ellos de las carpetas regulares.

Echemos un vistazo al repositorio Git resultante:

```
PS> git log --oneline --graph --decorate --all
* 44cd729 (tfs/featureA, featureA) Goodbye
* d202b53 Branched from $/tfvc-test/Trunk
* c403405 (HEAD, tfs/default, master) Hello
* b75da1a New project
PS> git log -1
commit c403405f4989d73a2c3c119e79021cb2104ce44a
Author: Ben Straub <ben@straub.cc>
Date:   Fri Aug 1 03:41:59 2014 +0000

    Hello

    git-tfs-id:
[https://username.visualstudio.com/DefaultCollection]$/myproject/Trunk;C16
```

Hay dos ramas locales, `master` y` featuresS`, que representan el punto inicial inicial del clon (`Trunk` en TFVC) y una rama secundaria (` featureS` en TFVC). También puede ver que el `tfs` remote '' también tiene un par de referencias:` default` y `featureA`, que representan las ramas de TFVC. Git-tfs mapea la rama desde la que clonaste a `tfs / default`, y otras obtienen sus propios nombres.

Otra cosa a notar es las líneas `git-tfs-id:` en los mensajes de confirmación. En lugar de etiquetas, git-tfs usa estos marcadores para relacionar los conjuntos de cambios de TFVC con las confirmaciones de Git. Esto tiene la consecuencia de que tus confirmaciones de Git tendrán un hash SHA-1 diferente antes y después de que se hayan enviado a TFVC.

Git-tf [s] Flujo de trabajo

NOTE

Independientemente de la herramienta que esté utilizando, debe establecer un par de valores de configuración de Git para evitar problemas.

```
$ git config set --local core.ignorecase=true
$ git config set --local core.autocrlf=false
```

Lo siguiente obvio que querrás hacer es trabajar en el proyecto. TFVC y TFS tienen varias características que pueden agregar complejidad a su flujo de trabajo:

1. Las ramas de características que no están representadas en TFVC agregan un poco de complejidad. Esto tiene que ver con las * muy * diferentes formas en que TFVC y Git representan las ramas.
2. Tenga en cuenta que TFVC permite a los usuarios "verificar" los archivos del servidor, bloqueándolos para que nadie más pueda editarlos. Obviamente, esto no le impedirá editarlos en su repositorio local, pero podría interferir cuando llegue el momento de enviar sus cambios al servidor TFVC.
3. TFS tiene el concepto de comprobaciones "compuertas", donde un ciclo de prueba de compilación TFS debe completarse satisfactoriamente antes de permitir el registro. Utiliza la función ‘` shelve ” en TFVC, que no cubrimos en detalle aquí. Puede falsificar esto de forma

manual con git-tf, y git-tfs proporciona el comando `checkintool` que es sensible a la puerta.

En aras de la brevedad, lo que trataremos aquí es el camino feliz, qué pasos laterales o evita la mayoría de estos problemas.

Flujo de trabajo del: `git-tf`

Digamos que has hecho algo de trabajo, has hecho un par de confirmaciones de Git en `master` y estás listo para compartir tu progreso en el servidor de TFVC. Aquí está nuestro repositorio de Git:

```
$ git log --oneline --graph --decorate --all
* 4178a82 (HEAD, master) update code
* 9df2ae3 update readme
* d44b17a (tag: TFS_C35190, origin_tfs/tfs) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
          Team Project Creation Wizard
```

Queremos tomar la instantánea que está en la confirmación `4178a82` y subirla al servidor TFVC. Lo primero es lo primero: veamos si alguno de nuestros compañeros de equipo hizo algo desde la última vez que nos conectamos:

```
$ git tf fetch
Username: domain\user
Password:
Connecting to TFS...
Fetching $/myproject at latest changeset: 100%, done.
Downloaded changeset 35320 as commit 8ef06a8. Updated FETCH_HEAD.
$ git log --oneline --graph --decorate --all
* 8ef06a8 (tag: TFS_C35320, origin_tfs/tfs) just some text
| * 4178a82 (HEAD, master) update code
| * 9df2ae3 update readme
|/
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
          Team Project Creation Wizard
```

Parece que alguien más está trabajando, también, y ahora tenemos una historia divergente. Aquí es donde brilla el Git, pero tenemos dos opciones de cómo proceder:

1. Hacer una confirmación de fusión se siente natural como un usuario de Git (después de todo, eso es lo que hace `git pull`), y git-tf puede hacer esto por ti con un simple` git tf pull`. Tenga en cuenta, sin embargo, que TFVC no piensa de esta manera, y si empuja la fusión se compromete su historia comenzará a verse diferente en ambos lados, lo que puede ser confuso. Sin embargo, si planea enviar todos sus cambios como un solo conjunto de cambios, esta es probablemente la

opción más fácil.

2. Rebasing hace que nuestro historial de compromisos sea lineal, lo que significa que tenemos la opción de convertir cada una de nuestras confirmaciones de Git en un conjunto de cambios de TFVC. Como esto deja la mayoría de las opciones abiertas, le recomendamos que lo haga de esta manera; git-tf incluso te lo facilita con `git tf pull --rebase`.

La decisión es tuya: Para este ejemplo, vamos a rebasar:

```
$ git rebase FETCH_HEAD
First, rewinding head to replay your work on top of it...
Applying: update readme
Applying: update code
$ git log --oneline --graph --decorate --all
* 5a0e25e (HEAD, master) update code
* 6eb3eb5 update readme
* 8ef06a8 (tag: TFS_C35320, origin_tfs/tfs) just some text
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
          Team Project Creation Wizard
```

Ahora estamos listos para hacer una comprobación en el servidor de TFVC. Git-tf le da la opción de hacer un único conjunto de cambios que represente todos los cambios desde el último (`--shallow`, que es el predeterminado) y crear un nuevo conjunto de cambios para cada confirmación de Git (` --deep`). Para este ejemplo, crearemos un solo conjunto de cambios:

```
$ git tf checkin -m 'Updating readme and code'
Username: domain\user
Password:
Connecting to TFS...
Checking in to $/myproject: 100%, done.
Checked commit 5a0e25e in as changeset 35348
$ git log --oneline --graph --decorate --all
* 5a0e25e (HEAD, tag: TFS_C35348, origin_tfs/tfs, master) update code
* 6eb3eb5 update readme
* 8ef06a8 (tag: TFS_C35320) just some text
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
          Team Project Creation Wizard
```

Hay una nueva etiqueta `TFS_C35348`, que indica que TFVC está almacenando la misma instantánea exacta que la confirmación` 5a0e25e`. Es importante tener en cuenta que no todas las confirmaciones del Git deben tener una contraparte exacta en TFVC; el compromiso `6eb3eb5`, por ejemplo, no existe en ninguna parte del servidor.

Ese es el flujo de trabajo principal. Hay un par de otras consideraciones que querrás tener en cuenta:

- No hay ramificación. Git-tf solo puede crear repositorios Git de una rama TFVC a la vez.
- Colabora con TFVC o Git, pero no con ambos. Los diferentes clones de git-tf del mismo repositorio de TFVC pueden tener diferentes hash de confirmación de SHA-1, lo que provocará innumerables dolores de cabeza.
- Si el flujo de trabajo de su equipo incluye la colaboración en Git y la sincronización periódica con TFVC, solo conéctese a TFVC con uno de los repositorios de Git.

Flujo de trabajo: `git-tfs`

Veamos el mismo escenario usando git-tfs. Aquí están las nuevas confirmaciones que hemos realizado en la rama `master` en nuestro repositorio de Git:

```
PS> git log --oneline --graph --all --decorate
* c3bd3ae (HEAD, master) update code
* d85e5a2 update readme
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 (tfs/default) Hello
* b75da1a New project
```

Ahora veamos si alguien más ha hecho un trabajo mientras estábamos hackeando:

```
PS> git tfs fetch
C19 = aea74a0313de0a391940c999e51c5c15c381d91d
PS> git log --all --oneline --graph --decorate
* aea74a0 (tfs/default) update documentation
| * c3bd3ae (HEAD, master) update code
| * d85e5a2 update readme
|/
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

Sí, resulta que nuestro compañero de trabajo ha agregado un nuevo conjunto de cambios de TFVC, que se muestra como el nuevo compromiso `aea74a0`, y la rama remota` tfs / default` se ha movido.

Al igual que con git-tf, tenemos dos opciones fundamentales sobre cómo resolver esta historia divergente:

1. Rebase para preservar una historia lineal.
2. Fusiona para preservar lo que realmente sucedió.

En este caso, haremos un checkin *' profundo '*, donde cada confirmación del Git se convierte en un conjunto de cambios de TFVC, por lo que queremos volver a establecer la base.

```
PS> git rebase tfs/default
First, rewinding head to replay your work on top of it...
Applying: update readme
Applying: update code
PS> git log --all --oneline --graph --decorate
* 10a75ac (HEAD, master) update code
* 5cec4ab update readme
* aea74a0 (tfs/default) update documentation
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

Ahora estamos listos para completar nuestra contribución al registrar nuestro código en el servidor TFVC. Usaremos el comando `rcheckin` aquí para crear un conjunto de cambios TFVC para cada commit de Git en la ruta de HEAD a la primera rama remota` tfs` encontrada (el comando `checkin` solo crearía un conjunto de cambios, más o menos como aplastar a Git se compromete).

```
PS> git tfs rcheckin
Working with tfs remote: default
Fetching changes from TFS to minimize possibility of late conflict...
Starting checkin of 5cec4ab4 'update readme'
 add README.md
C20 = 71a5ddce274c19f8fdc322b4f165d93d89121017
Done with 5cec4ab4b213c354341f66c80cd650ab98dcf1ed, rebasing tail onto new TFS-
commit...
Rebase done successfully.
Starting checkin of b1bf0f99 'update code'
 edit .git\tfs\default\workspace\ConsoleApplication1/ConsoleApplication1/Program.cs
C21 = ff04e7c35dfbe6a8f94e782bf5e0031cee8d103b
Done with b1bf0f9977b2d48bad611ed4a03d3738df05ea5d, rebasing tail onto new TFS-
commit...
Rebase done successfully.
No more to rcheckin.
PS> git log --all --oneline --graph --decorate
* ff04e7c (HEAD, tfs/default, master) update code
* 71a5ddc update readme
* aea74a0 update documentation
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

Observe cómo después de cada registro exitoso en el servidor TFVC, git-tfs vuelve a basar el trabajo restante en lo que acaba de hacer. Esto se debe a que está agregando el campo `git-tfs-id` al final de los mensajes de confirmación, lo que cambia los valores hash SHA-1. Esto es exactamente como se diseñó, y no hay nada de qué preocuparse, pero debe saber que está sucediendo, especialmente si comparte Git commits con otros.

TFS tiene muchas características que se integran con su sistema de control de versiones, como elementos de trabajo, revisores designados, registros bloqueados, etc. Puede ser engorroso trabajar con estas características usando solo una herramienta de línea de comandos, pero afortunadamente git-tfs le permite iniciar una herramienta gráfica de registro muy fácilmente:

```
PS> git tfs checkintool
PS> git tfs ct
```

Se parece un poco a esto:

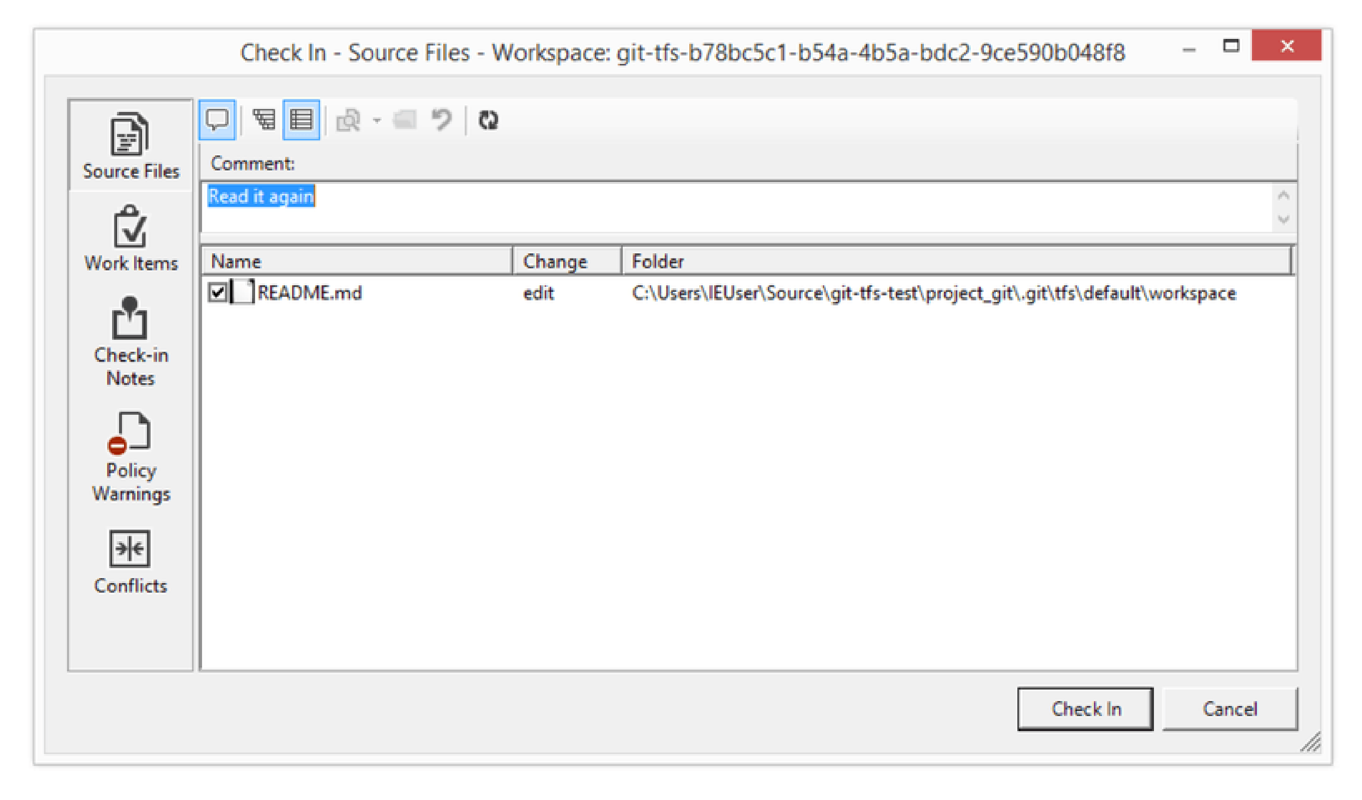


Figure 148. La herramienta de registro de git-tfs.

Esto resultará familiar para los usuarios de TFS, ya que es el mismo diálogo que se inicia desde Visual Studio.

Git-tfs también le permite controlar ramas de TFVC desde su repositorio de Git. Como ejemplo, creemos uno:

```
PS> git tfs branch $/tfvc-test/featureBee
The name of the local branch will be : featureBee
C26 = 1d54865c397608c004a2cadce7296f5edc22a7e5
PS> git log --oneline --graph --decorate --all
* 1d54865 (tfs/featureBee) Creation branch $/myproject/featureBee
* ff04e7c (HEAD, tfs/default, master) update code
* 71a5ddc update readme
* aea74a0 update documentation
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

Crear una rama en TFVC significa agregar un conjunto de cambios donde esa rama ahora existe, y esto se proyecta como una confirmación de Git. Tenga en cuenta también que git-tfs **created** la rama remota `tfs / featureBee`, pero` HEAD` todavía apunta a `master`. Si desea trabajar en la rama recién acuñada, querrá basar sus nuevas confirmaciones en la confirmación `1d54865`, tal vez creando una rama de tema a partir de esa confirmación.

Resumen de Git y TFS

Git-tf y Git-tfs son excelentes herramientas para interactuar con un servidor TFVC. Le permiten usar el poder de Git localmente, evitar tener que realizar un viaje de ida y vuelta al servidor central de TFVC, y hacer que su vida como desarrollador sea mucho más fácil, sin forzar a todo su equipo a migrar a Git. Si está trabajando en Windows (lo cual es probable si su equipo está usando TFS), probablemente quiera usar git-tfs, ya que su conjunto de características es más completo, pero si está trabajando en otra plataforma, ' Estaré usando git-tf, que es más limitado. Al igual que con la mayoría de las herramientas de este capítulo, debe elegir uno de estos sistemas de control de versiones para que sea canónico y usar el otro de forma subordinada: Git o TFVC deberían ser el centro de colaboración, pero no ambos.

Migración a Git

Si tiene una base de código existente en otro VCS pero ha decidido comenzar a usar Git, debe migrar su proyecto de una forma u otra. Esta sección revisa algunos importadores para sistemas comunes y luego demuestra cómo desarrollar su propio importador personalizado. Aprenderá a importar datos de varios de los sistemas SCM profesionales más grandes, ya que conforman la mayoría de los usuarios que están cambiando, y porque las herramientas de alta calidad para ellos son fáciles de conseguir.

Subversión

Si lee la sección anterior sobre el uso de git svn, usted puede usar fácilmente esas instrucciones para clonar un repositorio; luego, deje de usar el servidor de Subversión, presione en un nuevo servidor de Git y comience a usarlo. Si desea ver el historial, puede lograrlo tan rápido como pueda extraer los datos del servidor de Subversión (lo que puede llevar un tiempo).

Sin embargo, la importación no es perfecta; y porque tomará tanto tiempo, también puedes hacerlo bien. El primer problema es la información del autor. En Subversión, cada persona comprometida tiene un usuario en el sistema que está registrado en la información de confirmación. Los ejemplos en la sección anterior muestran schacon en algunos lugares, como la salida de culpa y el registro de git svn. Si desea asignar esto a mejores datos de autor de Git, necesita una asignación de los usuarios de Subversión a los autores de Git. Cree un archivo llamado users.txt que tenga esta asignación en un formato como este:

```
schacon = Scott Chacon <schacon@geemail.com>
selse = Someo Nelse <selse@geemail.com>
```

Para obtener una lista de los nombres de autor que utilizan SVN, puede ejecutar esto:

```
$ svn log --xml | grep author | sort -u | \
  perl -pe 's/.*>(.*?)<.*/$1 = /'
```

Eso genera la salida del registro en formato XML, luego mantiene solo las líneas con la información del autor, descarta los duplicados y elimina las etiquetas XML. (Obviamente, esto solo funciona en una máquina con grep, sort y perl instalados). Luego, redirija esa salida a su archivo users.txt para que pueda agregar los datos de usuario equivalentes de Git al lado de cada entrada.

Puede proporcionar este archivo a git svn para ayudarlo a mapear los datos del autor con mayor precisión. También puede indicarle a git svn que no incluya los metadatos que normalmente importa Subversión, pasando --no-metadata al comando clone o init. Esto hace que su comando de importación se vea así:

```
$ git svn clone http://my-project.googlecode.com/svn/ \
      --authors-file=users.txt --no-metadata -s my_project
```

Ahora debería tener una importación de Subversión más agradable en su directorio my_project. En lugar de commits que se ven así

```
commit 37efa680e8473b615de980fa935944215428a35a
Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
Date:   Sun May 3 00:12:22 2009 +0000

    fixed install - go to trunk

    git-svn-id: https://my-project.googlecode.com/svn/trunk@94 4c93b258-373f-11de-
    be05-5f7a86268029
```

se ven así:

```
commit 03a8785f44c8ea5cdb0e8834b7c8e6c469be2ff2
Author: Scott Chacon <schacon@geemail.com>
Date:   Sun May 3 00:12:22 2009 +0000

    fixed install - go to trunk
```

No solo el campo Autor se ve mucho mejor, sino que el git-svn-id ya no está allí.

También debería hacer un poco de limpieza posterior a la importación. Por un lado, debe limpiar las referencias raras que git svn configuró. Primero moverá las etiquetas para que sean etiquetas reales en lugar de ramas remotas extrañas, y luego moverá el resto de las ramas para que sean locales.

Para mover las etiquetas para que sean etiquetas Git correctas, ejecuta

```
$ cp -Rf .git/refs/remotes/origin/tags/* .git/refs/tags/
$ rm -Rf .git/refs/remotes/origin/tags
```

Esto toma las referencias que eran ramas remotas que comenzaron con controles remotos / origen / etiquetas / y las convierte en etiquetas reales (ligeras).

A continuación, mueva el resto de las referencias en refs / remotes para que sean sucursales locales:

```
$ cp -Rf .git/refs/remotes/* .git/refs/heads/
$ rm -Rf .git/refs/remotes
```

Ahora todas las ramas antiguas son ramas reales de Git y todas las etiquetas antiguas son etiquetas Git reales. Lo último que debe hacer es agregar su nuevo servidor Git como un control remoto y pulsarlo. Aquí hay un ejemplo de cómo agregar su servidor como un control remoto:

```
$ git remote add origin git@my-git-server:myrepository.git
```

Como quiere que todas sus ramas y etiquetas suban, ahora puede ejecutar esto:

```
$ git push origin --all
```

Todas sus ramas y etiquetas deben estar en su nuevo servidor Git en una importación agradable y limpia..

Mercurial

Since Mercurial and Git have fairly similar models for representing versions, and since Git is a bit more flexible, converting a repository from Mercurial to Git is fairly straightforward, using a tool

called "hg-fast-export", which you'll need a copy of:

```
$ git clone http://repo.or.cz/r/fast-export.git /tmp/fast-export
```

The first step in the conversion is to get a full clone of the Mercurial repository you want to convert:

```
$ hg clone <remote repo URL> /tmp/hg-repo
```

The next step is to create an author mapping file. Mercurial is a bit more forgiving than Git for what it will put in the author field for changesets, so this is a good time to clean house. Generating this is a one-line command in a `bash` shell:

```
$ cd /tmp/hg-repo
$ hg log | grep user: | sort | uniq | sed 's/user: *//' > ../authors
```

This will take a few seconds, depending on how long your project's history is, and afterwards the `/tmp/authors` file will look something like this:

```
bob
bob@localhost
bob <bob@company.com>
bob jones <bob <AT> company <DOT> com>
Bob Jones <bob@company.com>
Joe Smith <joe@company.com>
```

In this example, the same person (Bob) has created changesets under four different names, one of which actually looks correct, and one of which would be completely invalid for a Git commit. Hg-fast-export lets us fix this by adding `={new name and email address}` at the end of every line we want to change, and removing the lines for any usernames that we want to leave alone. If all the usernames look fine, we won't need this file at all. In this example, we want our file to look like this:

```
bob=Bob Jones <bob@company.com>
bob@localhost=Bob Jones <bob@company.com>
bob jones <bob <AT> company <DOT> com>=Bob Jones <bob@company.com>
bob <bob@company.com>=Bob Jones <bob@company.com>
```

The next step is to create our new Git repository, and run the export script:

```
$ git init /tmp/converted
$ cd /tmp/converted
$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A /tmp/authors
```

The `-r` flag tells hg-fast-export where to find the Mercurial repository we want to convert, and the

`-A` flag tells it where to find the author-mapping file. The script parses Mercurial changesets and converts them into a script for Git's "fast-import" feature (which we'll discuss in detail a bit later on). This takes a bit (though it's *much* faster than it would be over the network), and the output is fairly verbose:

```
$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A /tmp/authors
Loaded 4 authors
master: Exporting full revision 1/22208 with 13/0/0 added/changed/removed files
master: Exporting simple delta revision 2/22208 with 1/1/0 added/changed/removed files
master: Exporting simple delta revision 3/22208 with 0/1/0 added/changed/removed files
[…]
master: Exporting simple delta revision 22206/22208 with 0/4/0 added/changed/removed
files
master: Exporting simple delta revision 22207/22208 with 0/2/0 added/changed/removed
files
master: Exporting thorough delta revision 22208/22208 with 3/213/0
added/changed/removed files
Exporting tag [0.4c] at [hg r9] [git :10]
Exporting tag [0.4d] at [hg r16] [git :17]
[…]
Exporting tag [3.1-rc] at [hg r21926] [git :21927]
Exporting tag [3.1] at [hg r21973] [git :21974]
Issued 22315 commands
git-fast-import statistics:
---------------------------------------------------------------------
Alloc'd objects:     120000
Total objects:       115032 (    208171 duplicates                  )
      blobs  :        40504 (    205320 duplicates      26117 deltas of      39602
attempts)
      trees  :        52320 (      2851 duplicates      47467 deltas of      47599
attempts)
      commits:        22208 (         0 duplicates          0 deltas of          0
attempts)
      tags   :            0 (         0 duplicates          0 deltas of          0
attempts)
Total branches:         109 (         2 loads     )
      marks:        1048576 (     22208 unique    )
      atoms:           1952
Memory total:          7860 KiB
       pools:          2235 KiB
     objects:          5625 KiB
---------------------------------------------------------------------
pack_report: getpagesize()            =       4096
pack_report: core.packedGitWindowSize = 1073741824
pack_report: core.packedGitLimit      = 8589934592
pack_report: pack_used_ctr            =      90430
pack_report: pack_mmap_calls          =      46771
pack_report: pack_open_windows        =          1 /          1
pack_report: pack_mapped              =  340852700 /  340852700
---------------------------------------------------------------------

$ git shortlog -sn
   369  Bob Jones
   365  Joe Smith
```

That’s pretty much all there is to it. All of the Mercurial tags have been converted to Git tags, and Mercurial branches and bookmarks have been converted to Git branches. Now you’re ready to push the repository up to its new server-side home:

```
$ git remote add origin git@my-git-server:myrepository.git
$ git push origin --all
```

Perforce

El siguiente sistema que verá importando es Perforce Como mencionamos anteriormente, hay dos formas de que Git y Perforce hablen entre sí: git-p4 y Perforce Git Fusion.

Perforce Git Fusion

Git Fusion hace que este proceso sea bastante sencillo. Simplemente elija la configuración de su proyecto, las asignaciones de usuario y las ramas con un archivo de configuración (como se explica en << _ p4_git_fusion >>) y clone el repositorio. Git Fusion te deja con lo que parece ser un repositorio nativo de Git, que luego está listo para enviar a un host nativo de Git si lo deseas. Incluso puede usar Perforce como su host Git si gusta.

Git-p4

Git-p4 también puede actuar como una herramienta de importación. Como ejemplo, importaremos el proyecto Jam desde Perforce Public Depot. Para configurar su cliente, debe exportar la variable de entorno P4PORT para que se dirija al depósito de Perforce:

```
$ export P4PORT=public.perforce.com:1666
```

NOTE	Para poder seguir, necesitarás un depósito de Perforce para conectarte. Utilizaremos el depósito público en public.perforce.com para ver nuestros ejemplos, pero puede usar cualquier depósito al que tenga acceso.

Ejecute el comando `git p4 clone` para importar el proyecto Jam desde el servidor Perforce, proporcionando la ruta de depósito y proyecto y la ruta en la que desea importar el proyecto:

```
$ git-p4 clone //guest/perforce_software/jam@all p4import
Importación desde //guest/perforce_software/jam@all into p4import
Inicializó el repositorio vacío de Git en /private/tmp/p4import/.git/
Destino de importación: refs/remotes/p4/master
Importando revisión 9957 (100%)
```

Este proyecto en particular tiene solo una rama, pero esta configurado con vistas de ramificaciones (o simplemente un conjunto de directorios), y puede usar el indicador `--detect-branches` en` git p4 clone` para importar todas las ramas del proyecto. Ver << _ git_p4_branches >> para un poco más de detalle sobre esto.

En este punto, casi has terminado. Si va al directorio `p4import` y ejecuta` git log`, puede ver su trabajo importado:

```
$ git log -2
commit e5da1c909e5db3036475419f6379f2c73710c4e6
Autor: giles <giles@giles@perforce.com>
Información:   Wed Feb 8 03:13:27 2012 -0800

   Corrección a línea 355; change </UL> to </OL>.

    [git-p4: depot-paths = "//public/jam/src/": change = 8068]

commit aa21359a0a135dda85c50a7f7cf249e4f7b8fd98
Autor: kwirth <kwirth@perforce.com>
Información:   Tue Jul 7 01:35:51 2009 -0800

    Corrige el error de ortografía en la página Jam doc (cummulative -> cumulative).

    [git-p4: depot-paths = "//public/jam/src/": change = 7304]
```

Puede ver que `git-p4` ha dejado un identificador en cada mensaje de confirmación. Está bien mantener ese identificador allí, en caso de que necesite hacer referencia al número de cambio Perforce más adelante. Sin embargo, si desea eliminar el identificador, ahora es el momento de hacerlo, antes de comenzar a trabajar en el nuevo repositorio. Puede usar `git filter-branch` para eliminar las cadenas de identificador en masa:

```
$ git filter-branch --msg-filter 'sed -e "/^\[git-p4:/d"'
Rewrite e5da1c909e5db3036475419f6379f2c73710c4e6 (125/125)
Ref 'refs/heads/master' was rewritten
```

Si ejecuta `git log`, puede ver que todas las sumas de comprobación SHA-1 para las confirmaciones han cambiado, pero las cadenas` git-p4` ya no se encuentran en los mensajes de confirmación:

```
$ git log -2
commit b17341801ed838d97f7800a54a6f9b95750839b7
Autor: giles <giles@giles@perforce.com>
Información:   Wed Feb 8 03:13:27 2012 -0800

    Corrección a linea 355; change </UL> to </OL>.

commit 3e68c2e26cd89cb983eb52c024ecdfba1d6b3fff
Autor: kwirth <kwirth@perforce.com>
Información:   Tue Jul 7 01:35:51 2009 -0800

    Corrige el error de ortografía en la página Jam doc (cummulative -> cumulative).
```

Su importación está lista para subir a su nuevo servidor Git.

TFS

Si su equipo está convirtiendo su control de código fuente de TFVC a Git, querrá la conversión de mayor fidelidad que pueda obtener. Esto significa que mientras cubrimos git-tfs y git-tf para la sección de interoperabilidad, sólo cubriremos git-tfs para esta parte, porque git-tfs soporta ramificaciones, y esto es prohibitivamente difícil usando git-tf.

NOTE Se trata de una conversión unidireccional. El repositorio Git resultante no podrá conectarse con el proyecto TFVC original.

The first thing to do is map usernames. TFVC is fairly liberal with what goes into the author field for changesets, but Git wants a human-readable name and email address. You can get this information from the `tf` command-line client, like so:

```
PS> tf history $/myproject -recursive > AUTHORS_TMP
```

Esto agarra todos los conjuntos de cambios de la historia del proyecto y lo coloca en el archivo AUTHORS_TMP que procesaremos para extraer los datos de la columna *Usuario* (el segundo). Abre el archivo y busca en qué caracteres comienzan y terminan la columna y reemplazan, en la línea de comandos siguiente, los parámetros `11-20` del comando` cut` con los que se encuentran:

```
PS> cat AUTHORS_TMP | cut -b 11-20 | tail -n+3 | uniq | sort > AUTHORS
```

El comando `cut` mantiene sólo los caracteres entre 11 y 20 de cada línea. El comando `tail` omite las dos primeras líneas, que son cabeceras de campo y subrayados ASCII-art. El resultado de todo esto se canaliza a `uniq` para eliminar duplicados y se guarda en un archivo llamado` AUTHORS`. El siguiente paso es manual; Para que git-tfs haga un uso efectivo de este archivo, cada línea debe estar en este formato:

```
DOMAIN\username = User Name <email@address.com>
```

La parte de la izquierda es el campo "Usuario" de TFVC, y la porción en el lado derecho del signo de iguales es el nombre de usuario que se utilizará para los compromisos de Git.

Una vez que tengas este archivo, lo siguiente que debes hacer es hacer un clon completo del proyecto TFVC en el que estás interesado:

```
PS> git tfs clone --with-branches --authors=AUTHORS
https://username.visualstudio.com/DefaultCollection $/project/Trunk project_git
```

A continuación, deseará limpiar las secciones `git-tfs-id` desde la parte inferior de los mensajes de confirmación. El siguiente comando hará lo siguiente:

```
PS> git filter-branch -f --msg-filter 'sed "s/^git-tfs-id:.*$//g"' -- --all
```

Que utiliza el comando `sed` desde el entorno Git-bash para reemplazar cualquier línea que empiece por “git-tfs-id:” con vacío, que Git luego ignorará.

Una vez que todo está hecho, estás listo para añadir un nuevo mando a distancia, empujar todas sus ramas hacia arriba, y hacer que su equipo comience a trabajar desde Git.

Un proveedor individual

Si su sistema no es uno de los arriba mencionados, usted debe buscar un importador online - importadores de calidad están disponibles para muchos otros sistemas, incluido el CVS, Clear Case, Visual Source Safe, incluso un directorio de archivos. Si ninguna de estas herramientas funciona para usted, tiene una herramienta más obsoleta o necesita un método de importación más personalizado, debería usar ‘git fast-import’.. Este comando lee instrucciones simples de stdin para escribir datos Git específicos. Es mucho más fácil crear objetos Git de esta manera que ejecutar los comandos raw Git o intentar escribir los objetos raw (ver <Los entresijos internos de Git para más información). De esta manera, puede escribir un script de importación que lee la documentación necesaria del sistema desde el que está cargando e imprime instrucciones directas a stdout. A continuación, puede ejecutar este programa y canalizar su resultado mediante `git fast-import

Para demostrar rápidamente, podrá escribir un simple mensaje Supongamos que trabajas en `current`, haces una copia de seguridad de tu proyecto copiando el directorio ocasionalmente en un directorio de copia de seguridad con el código ‘back_YYYY_MM_DD" directorio de copia de seguridad y desea importar esta en Git. Tu Estructura del directorio se ve así:

```
$ ls /opt/import_from
back_2014_01_02
back_2014_01_04
back_2014_01_14
back_2014_02_03
current
```

Para poder importar un directorio Git, necesitas revisar cómo Git almacena sus datos. Como recordará, Git es fundamentalmente una listado enlazado de objetos commit que apuntan a una imágen de contenido. Todo lo que tienes por hacer es indicarle a `fast-import`qué son las snapshots de contenido, cuáles son los puntos de datos de confirmación y el orden en que van a entrar. Su estrategia será para ir a través de snapshots Uno a la vez y crear commits con el contenido de cada directorio, vinculando cada commit al anterior.

Como hicimos en Un ejemplo de implantación de una determinada política en Git, scribiremos esto en Ruby, porque es con lo que generalmente trabajamos y tiende a ser fácilmente legible.Usted puede escribir este ejemplo con bastante facilidad en cualquier elemento con el que esté familiarizado - sólo tiene que escribir la información apropiada a "stdout"., si se está ejecutando en Windows, esto significa que usted tendrá que tener especial cuidado de no introducir porte devoluciones al final de sus líneas - Importación rápida de git es muy particular acerca de que sólo quieren saltos de línea (LF) salto de línea (CRLF) usando windows.

Para empezar, cambiará al directorio de destino e identificará cada subdirectorio, cada uno de los cuales es snapshot que desea importar como un commit. Cambiará a cada subdirectorio e

imprimirán los comandos necesarios para exportarlo. Tu loop principal básico se ve así:

```
last_mark = nil

# loop through the directories
Dir.chdir(ARGV[0]) do
  Dir.glob("*").each do |dir|
    next if File.file?(dir)

    # move into the target directory
    Dir.chdir(dir) do
      last_mark = print_export(dir, last_mark)
    end
  end
end
```

Si ejecuta `print_export` dentro del directorio, que toma el registro y la señal de la instantánea anterior y devuelve el registro y la señal de ésta; de esa manera, se pueden enlazar correctamente.“Mark” is the `fast-import` para un código de identificación que se da a un commit; mientras crea commits,dar a cada uno de ellos una anotación que se puede utilizar para vincular al mismo desde otro commits. Por lo tanto, lo primero que debe hacer en el método ‘print_export’ es generar una etiqueta a partir del nombre del directorio:

```
mark = convert_dir_to_mark(dir)
```

Para ello, creará una matriz de directorios y utilizará el valor índice como punto, ya que una línea debe ser un código entero. Tu sistema se ve así:

```
$marks = []
def convert_dir_to_mark(dir)
  if !$marks.include?(dir)
    $marks << dir
  end
  ($marks.index(dir) + 1).to_s
end
```

Ahora que tiene una representación entera de su commit, necesita una dirección para los metadatos de commit. Como la información está expresada en el nombre del directorio, la analizarás. La siguiente línea del archivo `print_export` es

```
date = convert_dir_to_date(dir)
```

Donde`convert_dir_to_date` esta definido como

```
def convert_dir_to_date(dir)
  if dir == 'current'
    return Time.now().to_i
  else
    dir = dir.gsub('back_', '')
    (year, month, day) = dir.split('_')
    return Time.local(year, month, day).to_i
  end
end
```

Esto devuelve un valor entero para la data de cada directorio. La última parte de la meta-información que necesita para cada committer son los datos del committer, que usted codifica en una variable global:

```
$author = 'John Doe <john@example.com>'
```

Ahora está listo para empezar a publicar los datos de validación para su importador.: La información inicial indica que está definiendo un objeto de commit y en qué rama está activado, seguido de la línea que ha generado, la información de committer y el mensaje de committer y, a continuación, la línea anterior, si la hay. El código se ve así:

```
# print the import information
puts 'commit refs/heads/master'
puts 'mark :' + mark
puts "committer #{$author} #{date} -0700"
export_data('imported from ' + dir)
puts 'from :' + last_mark if last_mark
```

Usted codifica la zona horaria (-0700) porque hacerlo es fácil.

Si está importando desde otro sistema, debe especificar el uso horario como un intervalo de tiempo. El mensaje de confirmación debe expresarse en un formato especial:

```
data (size)\n(contents)
```

El formato consiste en los datos de la palabra, el tamaño de los datos a leer, una nueva línea, y finalmente los datos. Puesto que necesita utilizar el mismo formato para especificar el contenido del archivo más tarde, cree un sistema de ayuda,"export_data":[source,ruby]

```
def export_data(string)
  print "data #{string.size}\n#{string}"
end
```

Todo lo que queda es especificar el contenido del archivo para cada uno de ellos. snapshot. Esto es

fácil, porque tiene cada uno de ellos en un directorio - puede crear el comando `borrar todo` seguido del contenido de cada archivo en el directorio. Git entonces grabará cada instantánea apropiadamente:

```
puts 'deleteall'
Dir.glob("**/*").each do |file|
  next if !File.file?(file)
  inline_data(file)
end
```

Nota: Porque muchos sistemas piensan que sus modificaciones son cambios de una confirmación a otra, fast-import también puede tomar comandos con cada confirmación para especificar qué archivos se han agregado, eliminado o modificado y cuáles son los nuevos archivos. Podrías calcular las diferencias entre snapshots y proporcionar sólo estos archivos, pero hacerlo es más complejo - también puedes darle a Git todos los datos y dejar que lo resuelva. Si esto se adapta mejor a sus archivos, consulte la página de manual "Importación rápida" para obtener más detalles sobre cómo proporcionar sus archivos de esta manera. El formato para listar el nuevo contenido del archivo o especificar un archivo modificado con el nuevo contenido es el siguiente:

```
M 644 inline path/to/file
data (size)
(file contents)
```

Aquí, 644 es el modulo (si tiene archivos ejecutables, necesita detectar y especificar 755 en su lugar), e inline dice que listará el archivo inmediatamente después de esta línea. Su método de "datos_inline" se ve así:

```
def inline_data(file, code = 'M', mode = '644')
  content = File.read(file)
  puts "#{code} #{mode} inline #{file}"
  export_data(content)
end
```

Se reutiliza el modo `export_data` que definió anteriormente, porque es el mismo que el de los datos del mensaje de confirmación. Lo último que necesita hacer es devolver la línea actual para que pueda pasar a la siguiente edición:

```
return mark
```

NOTE Si se está ejecutando en Windows, deberá asegurarse de añadir un paso más. Como se ha dicho antes, Windows utiliza CRLF para nuevos carácteres de línea, mientras que git fast-import sólo espera LF. Para solucionar este problema y hacer git fast-importar feliz, necesita decirle a ruby para utilizar LF en lugar de CRLF:

```
$stdout.binmode
```

Eso es todo. Aquí está el script en su totalidad:

```
#!/usr/bin/env ruby

$stdout.binmode
$author = "John Doe <john@example.com>"

$marks = []
def convert_dir_to_mark(dir)
    if !$marks.include?(dir)
        $marks << dir
    end
    ($marks.index(dir)+1).to_s
end


def convert_dir_to_date(dir)
    if dir == 'current'
        return Time.now().to_i
    else
        dir = dir.gsub('back_', '')
        (year, month, day) = dir.split('_')
        return Time.local(year, month, day).to_i
    end
end

def export_data(string)
    print "data #{string.size}\n#{string}"
end

def inline_data(file, code='M', mode='644')
    content = File.read(file)
    puts "#{code} #{mode} inline #{file}"
    export_data(content)
end

def print_export(dir, last_mark)
    date = convert_dir_to_date(dir)
    mark = convert_dir_to_mark(dir)

    puts 'commit refs/heads/master'
```

```
    puts "mark :#{mark}"
    puts "committer #{$author} #{date} -0700"
    export_data("imported from #{dir}")
    puts "from :#{last_mark}" if last_mark

    puts 'deleteall'
    Dir.glob("**/*").each do |file|
        next if !File.file?(file)
        inline_data(file)
    end
    mark
end


# Loop through the directories
last_mark = nil
Dir.chdir(ARGV[0]) do
    Dir.glob("*").each do |dir|
        next if File.file?(dir)

        # move into the target directory
        Dir.chdir(dir) do
            last_mark = print_export(dir, last_mark)
        end
    end
end
```

If you run this script, you’ll get content that looks something like this:

```
$ ruby import.rb /opt/import_from
commit refs/heads/master
mark :1
committer John Doe <john@example.com> 1388649600 -0700
data 29
imported from back_2014_01_02deleteall
M 644 inline README.md
data 28
# Hello

This is my readme.
commit refs/heads/master
mark :2
committer John Doe <john@example.com> 1388822400 -0700
data 29
imported from back_2014_01_04from :1
deleteall
M 644 inline main.rb
data 34
#!/bin/env ruby

puts "Hey there"
M 644 inline README.md
(...)
```

Para ejecutar el importador, se debe pasar esta señal a través de `git fast-import` mientras que en el directorio Git se desea copiar. Puede crear un nuevo directorio y luego ejecutar `git init` en él para un punto de partida, y luego ejecutar su script:

```
$ git init
Initialized empty Git repository in /opt/import_to/.git/
$ ruby import.rb /opt/import_from | git fast-import
git-fast-import statistics:
---------------------------------------------------------------------
Alloc'd objects:       5000
Total objects:           13 (         6 duplicates                  )
      blobs  :            5 (         4 duplicates          3 deltas of          5
attempts)
      trees  :            4 (         1 duplicates          0 deltas of          4
attempts)
      commits:            4 (         1 duplicates          0 deltas of          0
attempts)
      tags   :            0 (         0 duplicates          0 deltas of          0
attempts)
Total branches:           1 (         1 loads     )
      marks:           1024 (         5 unique    )
      atoms:              2
Memory total:          2344 KiB
       pools:          2110 KiB
     objects:           234 KiB
---------------------------------------------------------------------
pack_report: getpagesize()            =       4096
pack_report: core.packedGitWindowSize = 1073741824
pack_report: core.packedGitLimit      = 8589934592
pack_report: pack_used_ctr            =         10
pack_report: pack_mmap_calls          =          5
pack_report: pack_open_windows        =          2 /          2
pack_report: pack_mapped              =       1457 /       1457
---------------------------------------------------------------------
```

Como puedes ver, cuando se completa con éxito, te da un conjunto de estadísticas sobre lo que logró. En este caso, ha importado un total de 13 objetos para 4 confirmaciones en 1 rama. Ahora, puedes ejecutar `git log` para ver tu nueva historial:

```
$ git log -2
commit 3caa046d4aac682a55867132ccdfbe0d3fdee498
Author: John Doe <john@example.com>
Date:   Tue Jul 29 19:39:04 2014 -0700

    imported from current

commit 4afc2b945d0d3c8cd00556fbe2e8224569dc9def
Author: John Doe <john@example.com>
Date:   Mon Feb 3 01:00:00 2014 -0700

    imported from back_2014_02_03
```

Ahí tienes, un buen y limpio depósito de Git. Es importante tener en cuenta que nada está comprobado - no tiene archivos en su directorio de trabajo al principio. Para obtenerlos, debe reajustar su rama a donde "maestro" está ahora:

```
$ ls
$ git reset --hard master
HEAD is now at 3caa046 imported from current
$ ls
README.md main.rb
```

Puedes hacer mucho más con la herramienta "Importación rápida": maneja diferentes modos, datos binarios, múltiples ramas y fusiones, etiquetas, indicadores de progreso y mucho más. Varios ejemplos de escenarios más complejos están disponibles en el directorio ‘contrib/fast-import’ del código fuente Git.

Resumen

Debería sentirse cómodo al usar Git como cliente para otros sistemas de control de versiones, o importar casi cualquier repositorio existente en Git sin perder datos. En el próximo capítulo, cubriremos los elementos internos de Git para que pueda crear cada byte, si es necesario.

Los entresijos internos de Git

Puede que hayas llegado a este capítulo saltando desde alguno previo o puede que hayas llegado tras leer todo el resto del libro - en uno u otro caso, aquí es donde aprenderás acerca del funcionamiento interno y la implementación de Git. Nos parece que esta información es realmente importante para entender cuan útil y potente es Git, pero algunas personas opinan que puede ser confuso e innecesariamente complejo para novatos. Por ello, lo hemos puesto en el capítulo final del libro; de tal forma que puedas leerlo antes o después, en cualquier momento, a lo largo de tu proceso de aprendizaje. Lo dejamos en tus manos.

Y, ahora que estamos aquí, comencemos con el tema. Ante todo, si no está aún suficientemente claro, Git es fundamentalmente un sistema de archivo de contenido localizable con una interfaz de usuario de VCS escrita sobre él. En breve vas a aprender más acerca de que significa esto.

En los primeros tiempos de Git (principalmente antes de la versión 1.5), la interfaz de usuario era mucho más compleja, ya que se centraba en el sistema de archivos en lugar de en mejorado VCS. En los últimos años, la IU se ha refinado hasta llegar a ser tan limpia y sencillo de usar como la de cualquier otro sistema; pero frecuentemente, el estereotipo persiste en lo complejo y difícil de aprender que era la IU anterior de Git.

La capa de contenido localizable del sistema de archivos es increíblemente interesante; por ello, es lo primero que vamos a cubrir en este capítulo. A continuación mostraremos los mecanismos de transporte y las tareas de mantenimiento del repositorio que posiblemente necesites usar alguna vez.

Fontanería y porcelana

Este libro habla acerca de como utilizar Git con más o menos 30 verbos, tales como `checkout`, `branch`, `remote`, etc. Pero, debido al origen de Git como una caja de herramientas para un VCS en lugar de como un completo y amigable sistema VCS, existen unos cuantos verbos para realizar tareas de bajo nivel y que se diseñaron para poder ser utilizados de forma encadenada al estilo UNIX o para ser utilizados en scripts. Estos comandos son conocidos como los "comandos de fontanería", mientras que los comandos más amigables son conocidos como los "comandos de porcelana".

Los primeros nueve capítulos de este libro se encargan casi exclusivamente de los comandos de porcelana. Pero en este capítulo trataremos sobre todo con los comandos de fontanería; comandos que te darán acceso a los entresijos internos de Git y que te ayudarán a comprender cómo y por qué hace Git lo que hace como lo hace. Muchos de estos comando no están pensados para ser utilizados manualmente desde la línea de comandos; sino más bien para ser utilizados como bloques de construcción para nuevas herramientas y scripts de usuario personalizados.

Cuando lanzas `git init` sobre una carpeta nueva o sobre una ya existente, Git crea la carpeta auxiliar `.git`; la carpeta donde se ubica prácticamente todo lo almacenado y manipulado por Git. Si deseas hacer una copia de seguridad de tu repositorio, con tan solo copiar esta carpeta a cualquier otro lugar ya tienes tu copia completa. Todo este capítulo se encarga de repasar el contenido en dicha carpeta. Tiene un aspecto como este:

```
$ ls -F1
HEAD
config*
description
hooks/
info/
objects/
refs/
```

Puede que veas algunos otros archivos en tu carpeta `.git`, pero este es el contenido de un repositorio recién creado tras ejecutar `git init`, -es la estructura por defecto. El archivo `description` se utiliza solo en el programa GitWeb; por lo que no necesitas preocuparte por él. El archivo `config` contiene las opciones de configuración específicas de este proyecto concreto, y la carpeta `info` guarda un archivo global de exclusión con los patrones a ignorar además de los presentes en el archivo `.gitignore`. La carpeta `hooks` contiene tus scripts, tanto de la parte cliente como de la parte servidor, tal y como se ha visto a detalle en el Puntos de enganche en Git.

Esto nos deja con cuatro entradas importantes: los archivos `HEAD` e `index` (todavía por ser creado), y las carpetas `objects` y `refs`. Estos elementos forman el núcleo de Git. La carpeta `objects` guarda el contenido de tu base de datos, la carpeta `refs` guarda los apuntadores a las confirmaciones de cambios (commits), el archivo `HEAD` apunta a la rama que tengas activa (checked out) en este momento, y el archivo `index` es donde Git almacena la información sobre tu área de preparación (staging área). Vamos a mirar en detalle cada una de esas secciones, para ver cómo trabaja Git.

Los objetos Git

Git es un sistema de archivo orientado a contenidos. Estupendo. Y eso, ¿qué significa? Pues significa que el núcleo Git es un simple almacén de claves y valores. Cuando insertas cualquier tipo de contenido, él te devuelve una clave que puedes utilizar para recuperar de nuevo dicho contenido en cualquier momento. Para verlo en acción, puedes utilizar el comando de fontanería *hash-object*. Este comando coge ciertos datos, los guarda en la carpeta *.git.* y te devuelve la clave bajo la cual se han guardado. Para empezar, inicializa un nuevo repositorio Git y comprueba que la carpeta *objects* está vacía.

```
$ git init test
Initialized empty Git repository in /tmp/test/.git/
$ cd test
$ find .git/objects
.git/objects
.git/objects/info
.git/objects/pack
$ find .git/objects -type f
```

Git ha inicializado la carpeta `objects`, creando en ella las subcarpetas `pack` e `info`; pero aún no hay archivos en ellas. Luego, guarda algo de texto en la base de datos de Git:

```
$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

La opción `-w` indica a `hash-object` que ha de guardar el objeto; de otro modo, el comando solo te respondería cual sería su clave. La opción `--stdin` indica al comando de leer desde la entrada estándar stdin; si no lo indicas, `hash-object` espera una ruta de archivo al final. La salida del comando es una suma de comprobación (checksum hash) de 40 caracteres. Este checksum hash SHA-1 es una suma de comprobación del contenido que estás guardando más una cabecera; cabecera sobre la que trataremos en breve. En estos momentos, ya puedes comprobar la forma en que Git ha guardado tus datos:

```
$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Como puedes ver, hay un archivo en la carpeta `objects`. En principio, esta es la forma en que guarda Git los contenidos; como un archivo por cada pieza de contenido, nombrado con la suma de comprobación SHA-1 del contenido y su cabecera. La subcarpeta se nombra con los primeros 2 caracteres del SHA-1, y el archivo con los restantes 38 caracteres.

Puedes volver a recuperar los contenidos usando el comando `cat-file`. Este comando es algo así como una "navaja suiza" para inspeccionar objetos Git. Pasándole la opción `-p`, puedes indicar al comando `cat-file` que deduzca el tipo de contenido y te lo muestre adecuadamente:

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
test content
```

Ahora que sabes cómo añadir contenido a Git y cómo recuperarlo de vuelta. Lo puedes hacer también con el propio contenido de los archivos. Por ejemplo, puedes realizar un control simple de versiones sobre un archivo. Para ello, crea un archivo nuevo y guarda su contenido en tu base de datos:

```
$ echo 'version 1' > test.txt
$ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
```

A continuación, escribe algo más de contenido en el archivo y vuélvelo a guardar:

```
$ echo 'version 2' > test.txt
$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

Tu base de datos contendrá las dos nuevas versiones del archivo, así como el primer contenido que habías guardado en ella antes:

```
$ find .git/objects -type f
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Podrás revertir el archivo a su primera versión

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt
$ cat test.txt
version 1
```

o a su segunda versión:

```
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt
$ cat test.txt
version 2
```

Pero no es práctico esto de andar recordando la clave SHA-1 para cada versión de tu archivo; es más, realmente no estás guardando el nombre de tu archivo en el sistema, --solo su contenido--. Este tipo de objeto se denomina un blob (binary large object). Con la orden `cat-file -t` puedes comprobar el tipo de cualquier objeto almacenado en Git, sin mas que indicar su clave SHA-1':

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
blob
```

Objetos tipo árbol

El siguiente tipo de objeto a revisar serán los árboles, que se encargan de resolver el problema de guardar un nombre de archivo, a la par que guardamos conjuntamente un grupo de archivos. Git guarda contenido de manera similar a un sistema de archivos UNIX, pero de forma algo más simple. Todo el contenido se guarda como objetos árbol (tree) u objetos binarios (blob), correspondiendo los árboles a las entradas de carpetas; y correspondiendo los binarios, mas o menos, a los contenidos de los archivos (inodes). Un objeto tipo árbol tiene una o más entradas de tipo árbol, cada una de las cuales consta de un puntero SHA-1 a un objeto binario (blob) o a un subárbol, más sus correspondientes datos de modo, tipo y nombre de archivo. Por ejemplo, el árbol más reciente en un proyecto puede ser algo como esto:

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859      README
100644 blob 8f94139338f9404f26296befa88755fc2598c289      Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0      lib
```

La sentencia `master^{tree}` indica el objeto árbol apuntado por la última confirmación de cambios (commit) en tu rama principal (master). Fíjate en que la carpeta `lib` no es un objeto binario, sino un

apuntador a otro objeto tipo árbol.

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b      simplegit.rb
```

Conceptualmente, la información almacenada por Git es algo similar a esto:

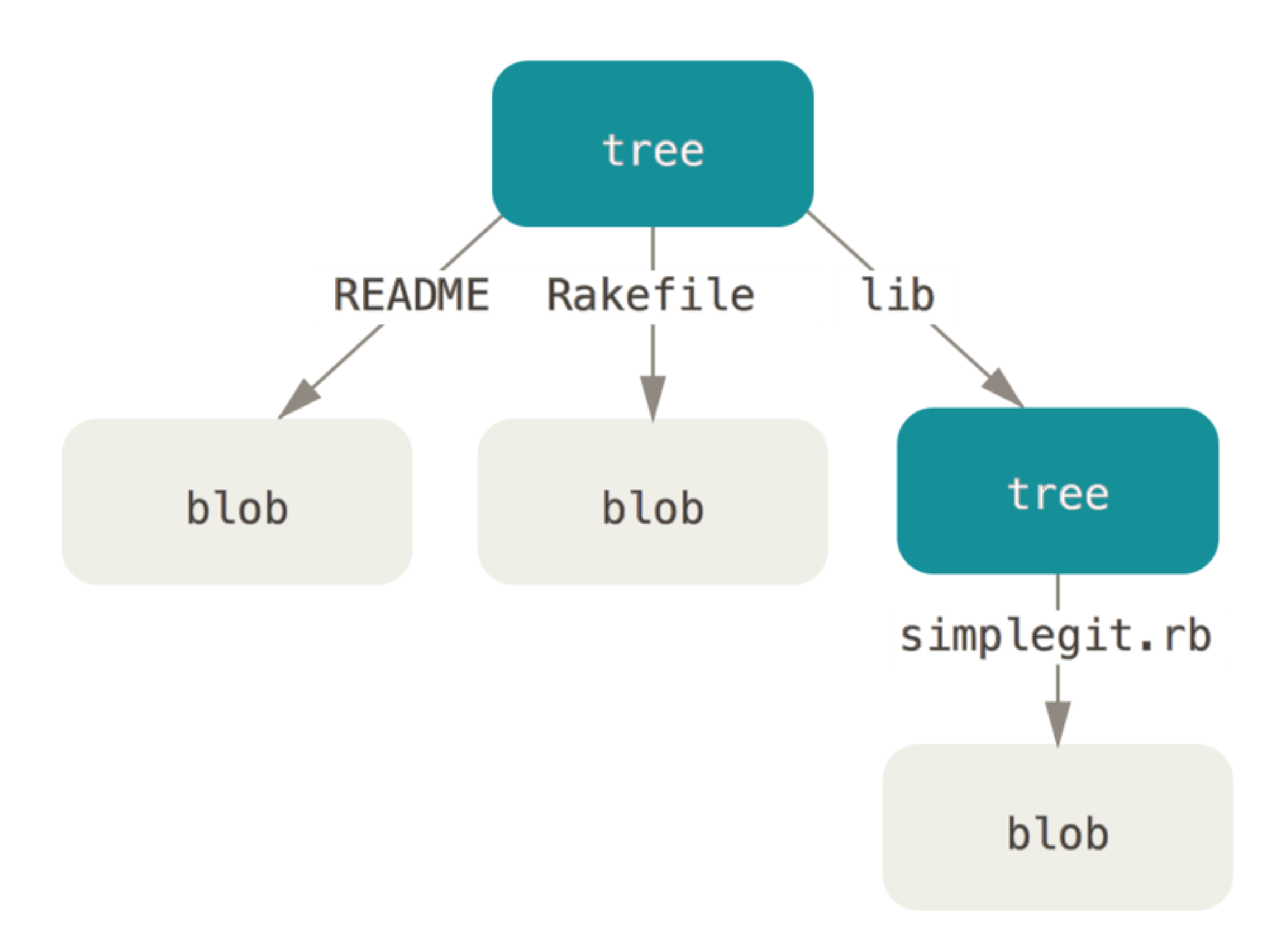


Figure 149. Versión simplificada del modelo de datos de Git.

Puedes crear fácilmente tu propio árbol. Habitualmente Git suele crear un árbol a partir del estado de tu área de preparación (staging area) o índice, escribiendo un serie de objetos árbol desde él. Por tanto, para crear un objeto árbol, previamente has de crear un índice preparando algunos archivos para ser almacenados. Puedes utilizar el comando de "fontanería" `update-index` para crear un índice con una sola entrada, --la primera versión de tu archivo text.txt--. Este comando se utiliza para añadir artificialmente la versión anterior del archivo test.txt a una nueva área de preparación. Has de utilizar la opción `--add`, porque el archivo no existe aún en tu área de preparación (es más, ni siquiera tienes un área de preparación), y has de utilizar también la opción `--cacheinfo`, porque el archivo que estás añadiendo no está en tu carpeta, sino en tu base de datos. Para terminar, has de indicar el modo, la clave SHA-1 y el nombre de archivo:

```
$ git update-index --add --cacheinfo 100644 \
  83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

En este caso, indicas un modo `100644`, el modo que denota un archivo normal. Otras opciones son `100755`, para un archivo ejecutable; o `120000`, para un enlace simbólico. Estos modos son como los modos de UNIX, pero mucho menos flexibles — solo estos tres modos son válidos para archivos (blobs) en Git; (aunque también se permiten otros modos para carpetas y submódulos) --.

Tras esto, puedes usar el comando `write-tree` para escribir el área de preparación a un objeto tipo árbol. Sin necesidad de la opción `-w`, solo llamando al comando `write-tree`, y si dicho árbol no existiera ya, se crea automáticamente un objeto tipo árbol a partir del estado del índice.

```
$ git write-tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579
100644 blob 83baae61804e65cc73a7201a7252750c76066a30      test.txt
```

También puedes comprobar si realmente es un objeto tipo árbol:

```
$ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579
tree
```

Vamos a crear un nuevo árbol con la segunda versión del archivo test.txt y con un nuevo archivo:

```
$ echo 'new file' > new.txt
$ git update-index test.txt
$ git update-index --add new.txt
```

El área de preparación contendrá ahora la nueva versión de test.txt, así como el nuevo archivo new.txt. Escribiendo este árbol, (guardando el estado del área de preparación o índice), podrás ver que aparece algo así como:

```
$ git write-tree
0155eb4229851634a0f03eb265b69f5a2d56f341
$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

Aquí se ven las entradas para los dos archivos y también el que la suma de comprobación SHA-1 de test.txt es la "segunda versión" de la anterior (`1f7a7a`). Simplemente por diversión, puedes añadir el primer árbol como una subcarpeta de este otro. Para leer árboles al área de preparación puedes utilizar el comando `read-tree`. Y, en este caso, puedes hacerlo como si fuera una subcarpeta utilizando la opción `--prefix`:

```
$ git read-tree --prefix=bak d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git write-tree
3c4e9cd789d88d8d89c1073707c3585e41b0e614
$ git cat-file -p 3c4e9cd789d88d8d89c1073707c3585e41b0e614
040000 tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579      bak
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

Si crearas una carpeta de trabajo a partir de este nuevo árbol que acabas de escribir, obtendrías los dos archivos en el nivel principal de la carpeta de trabajo y una subcarpeta llamada `bak` conteniendo la primera versión del archivo test.txt. Puedes pensar en algo parecido a esto para representar los datos guardados por Git para estas estructuras:

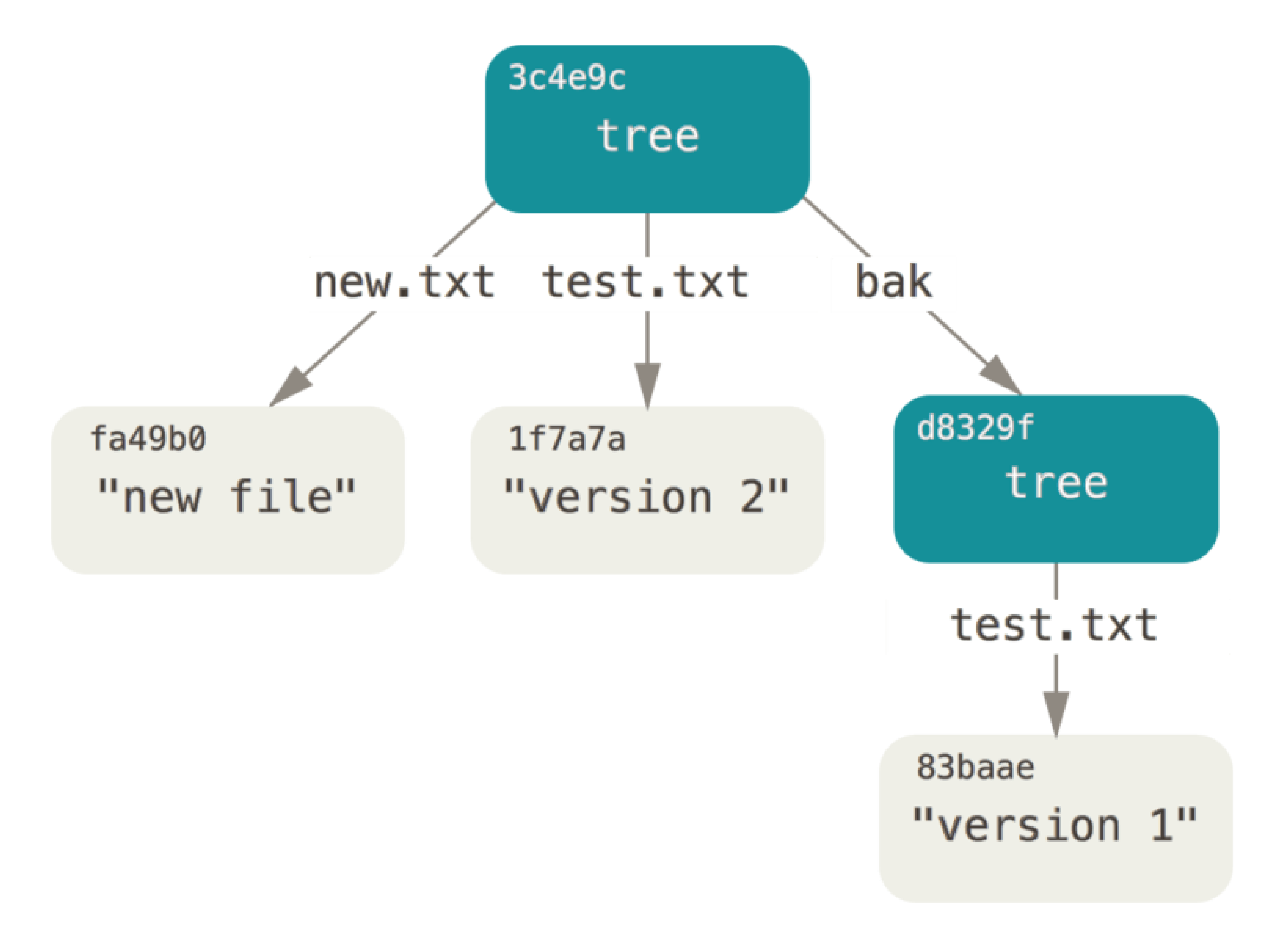


Figure 150. La estructura del contenido Git para tus datos actuales.

Objetos de confirmación de cambios

Tienes tres árboldes que representan diferentes momentos interesantes de tu proyecto, pero el problema principal sigue siendo el estar obligado a recordar los tres valores SHA-1 para poder recuperar cualquiera de esos momentos. Asimismo, careces de información alguna sobre quién guardó las instantáneas de esos momentos, cuándo fueron guardados o por qué se guardaron. Este es el tipo de información que almacenan para tí los objetos de confirmación de cambios.

Para crearlos, tan solo has de llamar al comando `commit-tree`, indicando uno de los árboles SHA-1 y

los objetos de confirmación de cambios que lo preceden (si es que lo precede alguno). Empezando por el primer árbol que has escrito:

```
$ echo 'first commit' | git commit-tree d8329f
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

Con el comando `cat-file` puedes revisar el nuevo objeto de confirmación de cambios recién creado:

```
$ git cat-file -p fdf4fc3
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
author Scott Chacon <schacon@gmail.com> 1243040974 -0700
committer Scott Chacon <schacon@gmail.com> 1243040974 -0700

first commit
```

El formato para un objeto de confirmación de cambios (commit) es sencillo, contemplando: el objeto tipo árbol para la situación del proyecto en ese momento puntual; la información sobre el autor/confirmador (que usa tus opciones de configuración `user.name` y `user.email` y una marca de tiempo); una línea en blanco; y el mensaje de la confirmación de cambios.

Puedes seguir adelante, escribiendo los otros dos objetos de confirmación de cambios, y relacionando cada uno de ellos con su inmediato anterior:

```
$ echo 'second commit' | git commit-tree 0155eb -p fdf4fc3
cac0cab538b970a37ea1e769cbbde608743bc96d
$ echo 'third commit'  | git commit-tree 3c4e9c -p cac0cab
1a410efbd13591db07496601ebc7a059dd55cfe9
```

Cada uno de estos tres objetos de confirmación de cambios apunta a uno de los tres objetos tipo árbol que habías creado previamente. Más aún, en estos momentos tienes ya un verdadero historial Git, que puedes comprobar con el comando `git log`, lanzándolo mientras estás en la última de las confirmaciones de cambio.

```
$ git log --stat 1a410e
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700

    third commit

 bak/test.txt | 1 +
 1 file changed, 1 insertion(+)

commit cac0cab538b970a37ea1e769cbbde608743bc96d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:14:29 2009 -0700

    second commit

 new.txt  | 1 +
 test.txt | 2 +-
 2 files changed, 2 insertions(+), 1 deletion(-)

commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:09:34 2009 -0700

    first commit

 test.txt | 1 +
 1 file changed, 1 insertion(+)
```

¡Sorprendente!. Acabas de confeccionar un historial Git utilizando solamente operaciones de bajo nivel, sin usar ninguno de los comandos de la interfaz principal. Esto es básicamente lo que hace Git cada vez que utilizas los comandos `git add` y `git commit`: guardar objetos binarios (blobs) para los archivos modificados, actualizar el índice, escribir árboles (trees), escribir objetos de confirmación de cambios (commits) que los referencian y relacionar cada uno de ellos con su inmediato precedente. Estos tres objetos Git, -binario, árbol y confirmación de cambios--, se guardan como archivos separados en la carpeta `.git/objects`. Aquí se muestran todos los objetos presentes en este momento en la carpeta del ejemplo, con comentarios acerca de lo que almacena cada uno de ellos:

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Siguiendo todos los enlaces internos, puedes llegar a un gráfico similar a esto:

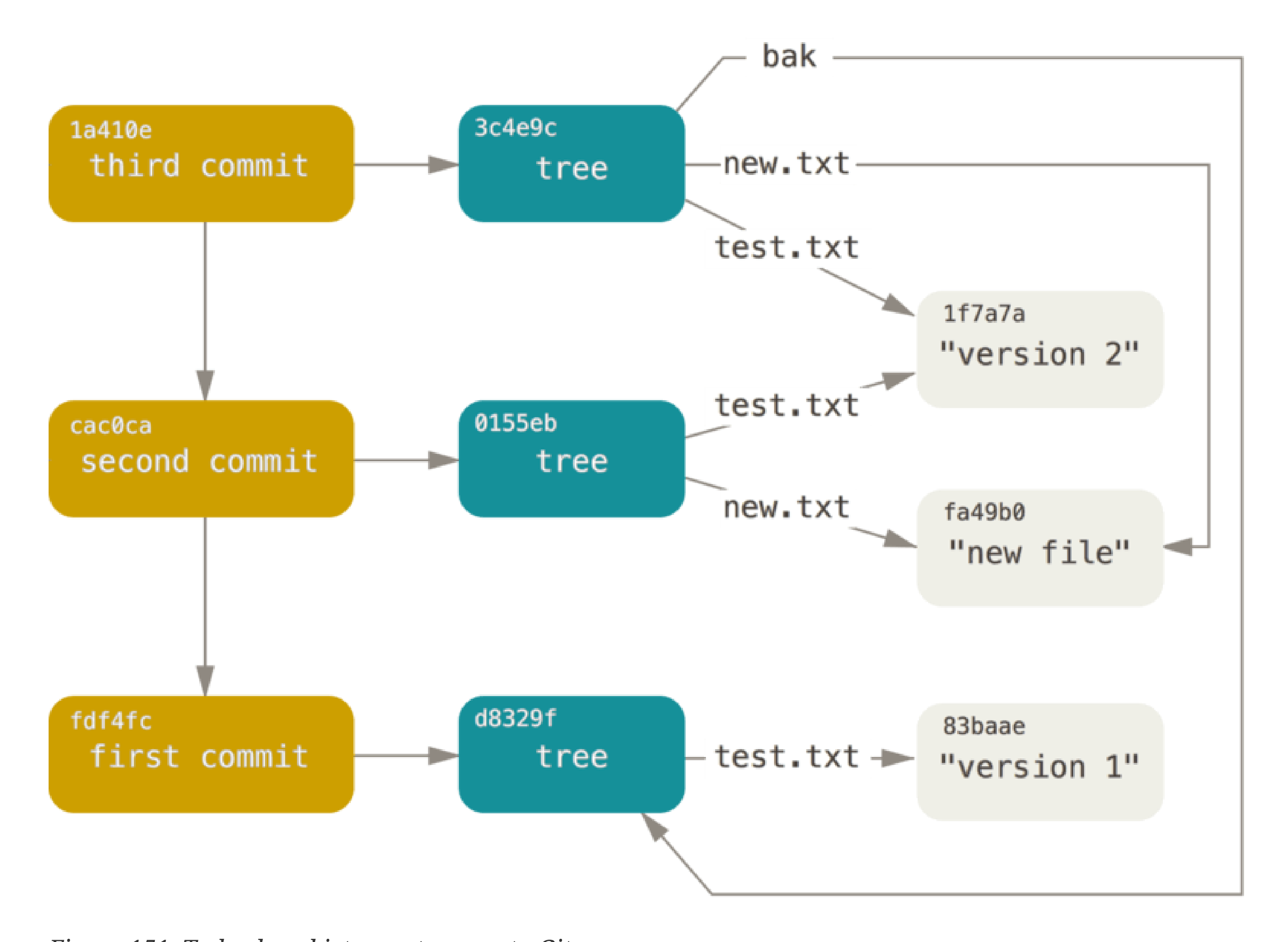


Figure 151. Todos los objetos en tu carpeta Git.

Almacenamiento de los objetos

Hemos citado anteriormente que siempre se almacena una cabecera junto al contenido. Vamos a echar un vistazo a cómo Git almacena sus objetos. Veamos el proceso de guardar un objeto binario grande (blob), --en este caso la cadena de texto "what is up, doc?" (¿qué hay de nuevo, viejo?)--, interactivamente, en el lenguaje de script Ruby.

Puedes arrancar el modo interactivo de Ruby con el comando `irb`:

```
$ irb
>> content = "what is up, doc?"
=> "what is up, doc?"
```

Git construye la cabecera comenzando por el tipo de objeto, en este caso un objeto binario grande (blob). Después añade un espacio, seguido del tamaño del contenido y termina con un byte nulo:

```
>> header = "blob #{content.length}\0"
=> "blob 16\u0000"
```

Git concatena la cabecera y el contenido original, para calcular la suma de control SHA-1 conjunta. En Ruby, para calcular el valor SHA-1 de una cadena de texto has de incluir la libreria de generación SHA1 con el comando `require` y llamar luego a la orden `Digest::SHA1.hexdigest()`:

```
>> store = header + content
=> "blob 16\u0000what is up, doc?"
>> require 'digest/sha1'
=> true
>> sha1 = Digest::SHA1.hexdigest(store)
=> "bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

Git comprime todo el contenido con zlib, lo cual puedes hacer en Ruby con la libreria zlib. Primero has de incluir la libreria y luego lanzar la orden `Zlib::Deflate.deflate()` sobre el contenido:

```
>> require 'zlib'
=> true
>> zlib_content = Zlib::Deflate.deflate(store)
=> "x\x9CK\xCA\xC9OR04c(\xCFH,Q\xC8,V(-\xD0QH\xC9O\xB6\a\x00_\x1C\a\x9D"
```

Para terminar, has de escribir el contenido comprimido en un objeto en disco. Para fijar el lugar donde almacenarlo, utilizaremos como nombre de carpeta los dos primeros caracteres del valor SHA-1 y como nombre de archivo los restantes 38 caracteres de dicho valor SHA-1. En Ruby, puedes utilizar la función `FileUtils.mkdir_p()` para crear una carpeta. Después, puedes abrir un archivo con la orden `File.open()` y escribir contenido en él con la orden `write()`:

```
>> path = '.git/objects/' + sha1[0,2] + '/' + sha1[2,38]
=> ".git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37"
>> require 'fileutils'
=> true
>> FileUtils.mkdir_p(File.dirname(path))
=> ".git/objects/bd"
>> File.open(path, 'w') { |f| f.write zlib_content }
=> 32
```

Y ¡esto es todo! --acabas de crear un auténtico objeto Git binario grande (blob)--. Todos los demas objetos Git se almacenan de forma similar, pero con la diferencia de que sus cabeceras comienzan con un tipo diferente — en lugar de *blob* (objeto binario grande), comenzarán por *commit* (confirmación de cambios), o por *tree* (árbol) --. Además, el contenido de un binario (blob) puede ser prácticamente cualquier cosa, mientras que el contenido de una confirmación de cambios (commit) o de un árbol (tree) han de seguir unos formatos internos muy concretos.

Referencias Git

Puedes utilizar algo así como `git log 1a410e` para echar un vistazo a lo largo de toda tu historia, recorriéndola y encontrando todos tus objetos. Pero para ello has necesitado recordar que la última confirmación de cambios es `1a410e`. Necesitarías un archivo donde almacenar los valores de las sumas de comprobación SHA-1, junto con sus respectivos nombres simples que puedas utilizar como enlaces en lugar de la propia suma de comprobación.

En Git, esto es lo que se conoce como "referencias" o "refs"; en la carpeta `.git/refs` puedes encontrar esos archivos con valores SHA-1 y nombres . En el proyecto actual, la carpeta aún no contiene archivos, pero sí contiene una estructura simple:

```
$ find .git/refs
.git/refs
.git/refs/heads
.git/refs/tags
$ find .git/refs -type f
```

Para crear una nueva referencia que te sirva de ayuda para recordar cual es tu última confirmación de cambios, puedes realizar técnicamente algo tan simple como:

```
$ echo "1a410efbd13591db07496601ebc7a059dd55cfe9" > .git/refs/heads/master
```

A partir de ese momento, puedes utilizar esa referencia principal que acabas de crear, en lugar del valor SHA-1, en todos tus comandos:

```
$ git log --pretty=oneline  master
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

No es conveniente editar directamente los archivos de referencia. Git suministra un comando mucho más seguro para hacer esto. Si necesitas actualizar una referencia, puedes utilizar el comando `update-ref`:

```
$ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
```

Esto es lo que es básicamente una rama en Git: un simple apuntador o referencia a la cabeza de una

línea de trabajo. Para crear una rama hacia la segunda confirmación de cambios, puedes hacer:

```
$ git update-ref refs/heads/test cac0ca
```

Y la rama contendrá únicamente trabajo desde esa confirmación de cambios hacia atrás.

```
$ git log --pretty=oneline test
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

En estos momentos, tu base de datos Git se parecerá conceptualmente a esto:

Figure 152. Objetos en la carpeta Git, con referencias a las cabeceras de las ramas.

Cuando lanzas comandos como `git branch (nombrederama)`, lo que hace Git es añadir, a cualquier nueva referencia que vayas a crear, el valor SHA-1 de la última confirmación de cambios en esa rama.

La CABEZA (HEAD)

Y ahora nos preguntamos, al lanzar el comando `git branch (nombrederama)`, ¿cómo sabe Git cuál es el valor SHA-1 de la última confirmación de cambios?. La respuesta a esta pregunta es el archivo HEAD (CABEZA).

El archivo HEAD es una referencia simbólica a la rama donde te encuentras en cada momento. Por referencia simbólica nos referimos a que, a diferencia de una referencia normal, esta contiene un enlace a otra referencia en lugar de un valor SHA-1. Si miras dentro del archivo, podrás observar algo como:

```
$ cat .git/HEAD
ref: refs/heads/master
```

Si lanzas el comando `git checkout test`, Git actualiza el contenido del archivo:

```
$ cat .git/HEAD
ref: refs/heads/test
```

Cuando lanzas una orden `git commit`, se crea un nuevo objeto de confirmación de cambios teniendo como padre la confirmación con valor SHA-1 a la que en ese momento esté apuntando la referencia en HEAD.

Puedes editar manualmente este archivo, pero, también para esta tarea existe un comando más seguro: `symbolic-ref`. Puedes leer el valor de HEAD a través de él:

```
$ git symbolic-ref HEAD
refs/heads/master
```

Y también puedes cambiar el valor de HEAD a través de él:

```
$ git symbolic-ref HEAD refs/heads/test
$ cat .git/HEAD
ref: refs/heads/test
```

Pero no puedes fijar una referencia simbólica fuera de "refs":

```
$ git symbolic-ref HEAD test
fatal: Refusing to point HEAD outside of refs/
```

Etiquetas

Acabamos de conocer los tres principales tipos de objetos Git, pero hay un cuarto. El objeto tipo etiqueta es muy parecido al tipo confirmación de cambios, --contiene un marcador, una fecha, un mensaje y un enlace--. Su principal diferencia reside en que generalmente apunta a una confirmación de cambios (commit) en lugar de a un árbol (tree). Es como una referencia a una rama, pero permaneciendo siempre inmóvil, --apuntando siempre a la misma confirmación de cambios--, dando un nombre mas amigable a esta.

Tal y como se ha comentado en Fundamentos de Git, hay dos tipos de etiquetas: las anotativas y las ligeras. Puedes crear una etiqueta ligera lanzando un comando tal como:

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

Una etiqueta ligera es simplemente eso: una referencia que nunca se mueve. Sin embargo, una etiqueta anotativa es más compleja. Al crear una etiqueta anotativa, Git crea un objeto tipo etiqueta y luego escribe una referencia apuntando a él en lugar de apuntar directamente a una confirmación de cambios. Puedes comprobarlo creando una: (la opción `-a` indica que la etiqueta es anotativa)

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'test tag'
```

Este es el objeto SHA-1 creado:

```
$ cat .git/refs/tags/v1.1
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

Ahora, lanzando el comando `cat-file` para ese valor SHA-1:

```
$ git cat-file -p 9585191f37f7b0fb9444f35a9bf50de191beadc2
object 1a410efbd13591db07496601ebc7a059dd55cfe9
type commit
tag v1.1
tagger Scott Chacon <schacon@gmail.com> Sat May 23 16:48:58 2009 -0700

test tag
```

Merece destacar que el inicio del objeto apunta al SHA-1 de la confirmación de cambios recién etiquetada. Y también el que no ha sido necesario apuntar directamente a una confirmación de cambios; realmente puedes etiquetar cualquier tipo de objeto Git. Por ejemplo, en el código fuente de Git los gestores han añadido su clave GPG pública como un objeto binario (blob) y lo han etiquetado. Puedes ver esta clave pública ejecutando esto en un clon del repositorio Git:

```
$ git cat-file blob junio-gpg-pub
```

El kernel de Linux tiene también un objeto tipo etiqueta apuntando a un objeto que no es una confirmación de cambios (commit). La primera etiqueta que se creó es la que apunta al árbol (tree) inicial donde se importó el código fuente.

Sitios remotos

El tercer tipo de referencia que puedes observar es la referencia a un sitio remoto. Si añades un sitio remoto y envías algo a él, Git almacenará en dicho sitio remoto el último valor para cada rama presente en la carpeta `refs/remotes`. Por ejemplo, puedes añadir un sitio remoto denominado `origin` y enviar a él tu rama `master`:

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
$ git push origin master
Counting objects: 11, done.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (7/7), 716 bytes, done.
Total 7 (delta 2), reused 4 (delta 1)
To git@github.com:schacon/simplegit-progit.git
  a11bef0..ca82a6d  master -> master
```

Tras lo cual puedes confirmar cual era la rama `master` en el remoto `origin` la última vez que comunicaste con el servidor, comprobando el archivo `refs/remotes/origin/master`:

```
$ cat .git/refs/remotes/origin/master
ca82a6dff817ec66f44342007202690a93763949
```

Las referencias a sitios remotos son distintas de las ramas normales (referencias en `refs/heads`), principalmente porque se las considera de sólo lectura. Puedes hacer `git checkout` a una, pero Git no apuntará HEAD a ella, de modo que nunca la actualizarás con el comando `commit`. Git las utiliza

solamente como marcadores al último estado conocido de cada rama en cada servidor remoto declarado.

Archivos empaquetadores

Volviendo a los objetos en la base de datos de tu repositorio Git de pruebas. En este momento, tienes 11 objetos --4 binarios, 3 árboles, 3 confirmaciones de cambios y 1 etiqueta--.

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/95/85191f37f7b0fb9444f35a9bf50de191beadc2 # tag
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Git comprime todos esos archivos con zlib, por lo que ocupan más bien poco, entre todos suponen solamente 925 bytes. Puedes añadir algún otro archivo de gran contenido al repositorio y verás una interesante funcionalidad de Git. Para demostrarlo, añadiremos el archivo `repo.rb` de la librería Grit --es un archivo de código fuente de unos 22K--:.

```
$ curl https://raw.githubusercontent.com/mojombo/grit/master/lib/grit/repo.rb >
repo.rb
$ git add repo.rb
$ git commit -m 'added repo.rb'
[master 484a592] added repo.rb
 3 files changed, 709 insertions(+), 2 deletions(-)
 delete mode 100644 bak/test.txt
 create mode 100644 repo.rb
 rewrite test.txt (100%)
```

Si hechas un vistazo al árbol resultante, podrás observar el valor SHA-1 del objeto binario correspondiente a dicho archivo repo.rb:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

Puedes usar `git cat-file` para ver como de grande es este objeto:

```
$ git cat-file -s 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5
22044
```

Ahora, modifica un poco dicho archivo y comprueba lo que sucede:

```
$ echo '# testing' >> repo.rb
$ git commit -am 'modified repo a bit'
[master 2431da6] modified repo.rb a bit
 1 file changed, 1 insertion(+)
```

Revisando el árbol creado por esta última confirmación de cambios, verás algo interesante:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob b042a60ef7dff760008df33cee372b945b6e884e      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

El objeto binario es ahora un binario completamente diferente. Aunque solo has añadido una única línea al final de un archivo que ya contenía 400 líneas, Git ha almacenado el resultado como un objeto completamente nuevo.

```
$ git cat-file -s b042a60ef7dff760008df33cee372b945b6e884e
22054
```

Y, así, tienes en tu disco dos objetos de 22 Kbytes prácticamente idénticos. ¿No seria práctico si Git pudiera almacenar uno de ellos completo y luego solo las diferencias del segundo con respecto al primero?

Pues bien, Git lo puede hacer. El formato inicial como Git guarda sus objetos en disco es el formato conocido como "relajado" (loose). Pero, sin embargo, de vez en cuando, Git suele agrupar varios de esos objetos en un único binario denominado archivo "empaquetador", para ahorrar espacio y hacer así más eficiente su almacenamiento. Esto sucede cada vez que tiene demasiados objetos en formato "relajado"; o cuando tu invocas manualmente al comando `git gc`; o justo antes de enviar cualquier cosa a un servidor remoto. Puedes comprobar el proceso pidiéndole expresamente a Git que empaquete objetos, utilizando el comando `git gc`:

```
$ git gc
Counting objects: 18, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (18/18), done.
Total 18 (delta 3), reused 0 (delta 0)
```

Tras esto, si miras los objetos presentes en la carpeta, veras que han desaparecido la mayoría de los

que había anteriormente y han apareciendo un par de objetos nuevos:

```
$ find .git/objects -type f
.git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects/info/packs
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.idx
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack
```

Solo han quedado aquellos objetos binarios no referenciados por ninguna confirmación de cambios --en este caso, el ejemplo de "¿que hay de nuevo, viejo?" y el ejemplo de "contenido de pruebas". Porque nunca los has llegado a incluir en ninguna confirmación de cambios, no se han considerado como objetos definitivos y, por tanto, no han sido empaquetados.

Los otros archivos presentes son el nuevo archivo empaquetador y un índice. El archivo empaquetador es un único archivo conteniendo dentro de él todos los objetos sueltos eliminados del sistema de archivo. El índice es un archivo que contiene las posiciones de cada uno de esos objetos dentro del archivo empaquetador, permitiéndonos así buscarlos y extraer rápidamente cualquiera de ellos. Lo que es interesante es el hecho de que, aunque los objetos originales presentes en el disco antes del `gc` ocupaban unos 22 Kbytes, el nuevo archivo empaquetador apenas ocupa 7 Kbytes. Empaquetando los objetos, has conseguido reducir a el uso de disco.

¿Cómo consigue Git esto? Cuando Git empaqueta objetos, va buscando archivos de igual nombre y tamaño similar, almacenando únicamente las diferencias entre una versión de cada archivo y la siguiente. Puedes comprobarlo mirando en el interior del archivo empaquetador. Y, para eso, has de utilizar el comando "de fontanería" `git verify-pack`:

```
$ git verify-pack -v .git/objects/pack/pack-
978e03944f5c581011e6998cd0e9e30000905586.idx
2431da676938450a4d72e260db3bf7b0f587bbc1 commit 223 155 12
69bcdaff5328278ab1c0812ce0e07fa7d26a96d7 commit 214 152 167
80d02664cb23ed55b226516648c7ad5d0a3deb90 commit 214 145 319
43168a18b7613d1281e5560855a83eb8fde3d687 commit 213 146 464
092917823486a802e94d727c820a9024e14a1fc2 commit 214 146 610
702470739ce72005e2edff522fde85d52a65df9b commit 165 118 756
d368d0ac0678cbe6cce505be58126d3526706e54 tag    130 122 874
fe879577cb8cffcdf25441725141e310dd7d239b tree   136 136 996
d8329fc1cc938780ffdd9f94e0d364e0ea74f579 tree   36 46 1132
deef2e1b793907545e50a2ea2ddb5ba6c58c4506 tree   136 136 1178
d982c7cb2c2a972ee391a85da481fc1f9127a01d tree   6 17 1314 1 \
  deef2e1b793907545e50a2ea2ddb5ba6c58c4506
3c4e9cd789d88d8d89c1073707c3585e41b0e614 tree   8 19 1331 1 \
  deef2e1b793907545e50a2ea2ddb5ba6c58c4506
0155eb4229851634a0f03eb265b69f5a2d56f341 tree   71 76 1350
83baae61804e65cc73a7201a7252750c76066a30 blob   10 19 1426
fa49b077972391ad58037050f2a75f74e3671e92 blob   9 18 1445
b042a60ef7dff760008df33cee372b945b6e884e blob   22054 5799 1463
033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 blob   9 20 7262 1 \
  b042a60ef7dff760008df33cee372b945b6e884e
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a blob   10 19 7282
non delta: 15 objects
chain length = 1: 3 objects
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack: ok
```

Puedes observar que el objeto binario `033b4`, (correspondiente a la primera versión de tu archivo repo.rb), tiene una referencia al binario `b042a` (la segunda versión de ese archivo). La tercera columna refleja el tamaño de cada objeto dentro del paquete, observándose que `b042a` ocupa unos 22 Kbytes; pero `033b4` solo ocupa 9 bytes. Resulta curioso que se almacene completa la segunda versión del archivo, mientras que la versión original es donde se almacena solo la diferencia --esto se debe a la mayor probabilidad de que vayamos a recuperar rápidamente la versión mas reciente del archivo--.

Lo verdaderamente interesante de todo este proceso es que podemos reempaquetar en cualquier momento. De vez en cuando, Git, en su empeño por optimizar la ocupación de espacio, reempaqueta automáticamente toda la base de datos, pero también tu mismo puedes reempaquetar en cualquier momento, lanzando manualmente el comando `git gc`.

Las especificaciones para hacer referencia a... (refspec)

A lo largo del libro hemos utilizado sencillos mapeados entre ramas remotas y referencias locales, pero las cosas pueden ser bastante más complejas. Supón que añades un remoto tal que:

```
$ git remote add origin https://github.com/schacon/simplegit-progit
```

Esto añade una nueva sección a tu archivo .git/config, indicando el nombre del remoto (origin), la ubicación (URL) del repositorio remoto y la referencia para recuperar (fetch) desde él:

```
[remote "origin"]
    url = https://github.com/schacon/simplegit-progit
    fetch = +refs/heads/*:refs/remotes/origin/*
```

El formato para esta referencia es un signo + opcional, seguido de una sentencia <orig>:<dest>; donde <orig> es la plantilla para referencias en el lado remoto y <dest> el lugar donde esas referencias se escribirán en el lado local. El +, si está presente, indica a Git que debe actualizar la referencia incluso en los casos en que no se dé un avance rápido (fast-forward).

En el caso por defecto en que es escrito por un comando git remote add, Git recupera del servidor todas las referencias bajo refs/heads/, y las escribe localmente en refs/remotes/origin/. De tal forma que, si existe una rama master en el servidor, puedes acceder a ella localmente a través de

```
$ git log origin/master
$ git log remotes/origin/master
$ git log refs/remotes/origin/master
```

Todas estas sentencias son equivalentes, ya que Git expande cada una de ellas a refs/remotes/origin/master.

Si, en cambio, quisieras hacer que Git recupere únicamente la rama master y no cualquier otra rama en el servidor remoto, puedes cambiar la linea de recuperación a

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

Quedando así esta referencia como la referencia por defecto para el comando git fetch para ese remoto. Para hacerlo puntualmente en un momento concreto, puedes especificar la referencia directamente en la linea de comando. Para recuperar la rama master del servidor remoto a tu rama origin/mymaster local, puedes lanzar el comando

```
$ git fetch origin master:refs/remotes/origin/mymaster
```

Puedes incluso indicar múltiples referencias en un solo comando. Escribiendo algo así como:

```
$ git fetch origin master:refs/remotes/origin/mymaster \
     topic:refs/remotes/origin/topic
From git@github.com:schacon/simplegit
 ! [rejected]        master     -> origin/mymaster  (non fast forward)
 * [new branch]      topic      -> origin/topic
```

En este ejemplo, se ha rechazado la recuperación de la rama master porque no era una referencia de avance rápido (fast-forward). Puedes forzarlo indicando el signo `+` delante de la referencia.

Es posible asimismo indicar referencias múltiples en el archivo de configuración. Si, por ejemplo, siempre recuperas las ramas *master* y *experiment*, puedes poner dos líneas:

```
[remote "origin"]
    url = https://github.com/schacon/simplegit-progit
    fetch = +refs/heads/master:refs/remotes/origin/master
    fetch = +refs/heads/experiment:refs/remotes/origin/experiment
```

Pero, en ningún caso puedes poner referencias genéricas parciales; por ejemplo, algo como esto sería erroneo:

```
fetch = +refs/heads/qa*:refs/remotes/origin/qa*
```

Aunque, para conseguir algo similar, puedes utilizar los espacios de nombres . Si tienes un equipo QA que envía al servidor una serie de ramas, y deseas recuperar la rama master y cualquier otra de las ramas del equipo, pero no recuperar ninguna rama de otro equipo, puedes utilizar una sección de configuración como esta:

```
[remote "origin"]
    url = https://github.com/schacon/simplegit-progit
    fetch = +refs/heads/master:refs/remotes/origin/master
    fetch = +refs/heads/qa/*:refs/remotes/origin/qa/*
```

De esta forma, puedes asignar fácilmente espacios de nombres; y resolver así complejos flujos de trabajo donde tengas simultáneamente, por ejemplo, un equipo QA enviando ramas, varios desarrolladores enviando ramas también y equipos integradores enviando y colaborando en ramas remotas.

Enviando (push) referencias

Es útil poder recuperar (fetch) referencias relativas en espacios de nombres, tal y como hemos visto, pero, ¿cómo pueden enviar (push) sus ramas al espacio de nombres `qa/` los miembros de equipo QA ?. Pues utilizando las referencias (refspecs) para enviar.

Si alguien del equipo QA quiere enviar su rama `master` a la ubicación `qa/master` en el servidor remoto, puede lanzar algo así como:

```
$ git push origin master:refs/heads/qa/master
```

Y, para que se haga de forma automática cada vez que ejecute `git push origin`, puede añadir una entrada `push` a su archivo de configuración:

```
[remote "origin"]
    url = https://github.com/schacon/simplegit-progit
    fetch = +refs/heads/*:refs/remotes/origin/*
    push = refs/heads/master:refs/heads/qa/master
```

Esto hará que un simple comando `git push origin` envíe por defecto la rama local `master` a la rama remota `qa/master`,

Borrando referencias

Se pueden utilizar las referencias (refspec) para borrar en el servidor remoto. Por ejemplo, lanzando algo como:

```
$ git push origin :topic
```

Se elimina la rama *topic* del servidor remoto, ya que la sustituimos por nada. (Al ser la referencia <origen>:<destino>, si no indicamos la parte <origen>, realmente estamos diciendo que enviamos *nada* a <destino>.)

Protocolos de transferencia

Git puede transferir datos entre dos repositorios utilizando uno de sus dos principales mecanismos de transporte: sobre protocolo '*tonto*', o sobre protocolo '*inteligente*'. En esta parte, se verán sucintamente cómo trabajan esos dos tipos de protocolo.

El protocolo tonto

Si vas a configurar un repositorio para ser servido en forma de sólo lectura a través de HTTP, es probable que uses el protocolo tonto. Este protocolo se llama '*tonto*' porque no requiere ningún tipo de código Git en la parte servidor durante el proceso de transporte; el proceso de recuperación (fetch) de datos se limita a una serie de peticiones GET, siendo el cliente quien ha de conocer la estructura del repositorio Git en el servidor.

NOTE

El protocolo tonto es muy poco usado hoy en día. Es difícil dar confidencialidad, por lo que la mayoría de los servidores Git (tanto los basados en la nube como los normales) se negarán a usarlo. Por lo general se recomienda utilizar el protocolo inteligente, que se describe un poco más adelante.

Vamos a revisar el proceso `http-fetch` para una librería simple de Git:

```
$ git clone http://server/simplegit-progit.git
```

Lo primero que hace este comando es recuperar el archivo `info/refs`. Este es un archivo escrito por el comando `update-server-info`, el que has de habilitar como enganche (hook) `post-receive` para permitir funcionar correctamente al transporte HTTP:

```
=> GET info/refs
ca82a6dff817ec66f44342007202690a93763949     refs/heads/master
```

A partir de ahí, ya tienes una lista de las referencias remotas y sus SHA-1s. Lo siguiente es mirar cual es la referencia a HEAD, de tal forma que puedas saber el punto a activar (checkout) cuando termines:

```
=> GET HEAD
ref: refs/heads/master
```

Ves que es la rama `master` la que has de activar cuando el proceso esté completado. En este punto, ya estás preparado para seguir procesando el resto de los objetos. En el archivo `info/refs` se ve que el punto de partida es la confirmación de cambios (commit) `ca82a6`, y, por tanto, comenzaremos recuperándola:

```
=> GET objects/ca/82a6dff817ec66f44342007202690a93763949
(179 bytes of binary data)
```

Cuando recuperas un objeto, dicho objeto se encuentra suelto (loose) en el servidor y lo traes mediante una petición estática HTTP GET. Puedes descomprimirlo, quitarle la cabecera y mirar el contenido:

```
$ git cat-file -p ca82a6dff817ec66f44342007202690a93763949
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

changed the version number
```

Tras esto, ya tienes más objetos a recuperar --el árbol de contenido `cfda3b` al que apunta la confirmación de cambios; y la confirmación de cambios padre `085bb3`--.

```
=> GET objects/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
(179 bytes of data)
```

El siguiente objeto confirmación de cambio (commit). Y el árbol de contenido:

```
=> GET objects/cf/da3bf379e4f8dba8717dee55aab78aef7f4daf
(404 - Not Found)
```

Pero…¡Ay!…parece que el objeto árbol no está suelto en el servidor, por lo que obtienes una respuesta 404 (objeto no encontrado). Puede haber un par de razones para que suceda esto: el objeto está en otro repositorio alternativo; o el objeto está en este repositorio, pero dentro de un objeto empaquetador (packfile). Git comprueba primero a ver si en el listado hay alguna alternativa:

```
=> GET objects/info/http-alternates
(empty file)
```

En el caso de que esto devolviera una lista de ubicaciones (URL) alternativas, Git busca en ellas (es un mecanismo muy adecuado en aquellos proyectos donde hay segmentos derivados uno de otro compartiendo objetos en disco.) Pero, en este caso, no hay alternativas, por lo que el objeto debe encontrarse dentro de un empaquetado. Para ver que empaquetados hay disponibles en el servidor, has de recuperar el archivo `objects/info/packs`, que contiene una lista de todos ellos: (que ha sido generada por `update-server-info`)

```
=> GET objects/info/packs
P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

Vemos que hay un archivo empaquetado, y el objeto buscado ha de encontrarse dentro de él; pero merece comprobarlo revisando el archivo de índice, para asegurarse. Hacer la comprobación es sobre todo útil en aquellos casos donde existan múltiples archivos empaquetados en el servidor, para determinar así en cual de ellos se encuentra el objeto que necesitas:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx
(4k of binary data)
```

Una vez tengas el índice del empaquetado, puedes mirar si el objeto buscado está en él, (Dicho índice contiene la lista de SHA-1s de los objetos dentro del empaquetado y las ubicaciones -offsets- de cada uno de ellos dentro de él). Una vez comprobada la presencia del objeto, adelante con la recuperación de todo el archivo empaquetado:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
(13k of binary data)
```

Cuando tengas el objeto árbol, puedes continuar avanzando por las confirmaciones de cambio. Y, como éstas también están dentro del archivo empaquetado que acabas de descargar, ya no necesitas hacer mas peticiones al servidor. Git activa una copia de trabajo de la rama `master` señalada por la referencia HEAD que has descargado al principio.

El protocolo inteligente

El protocolo tonto es simple pero ineficiente, y no puede manejar la escritura de datos desde el cliente al servidor. El protocolo inteligente es un método mucho más común de transmisión de datos, pero requiere un proceso en el lado remoto que es inteligente acerca de Git --puede leer datos localmente, determinar lo que el cliente tiene y necesita, y generar un empaquetado expresamente para él--. Existen dos conjuntos de procesos para transferir datos: uno para enviar y otro para recibir.

Enviando datos (uploading)

Para enviar datos a un proceso remoto, Git utliza `send-pack` (enviar paquete) y `receive-pack` (recibir paquete). El proceso `send-pack` corre en el cliente y conecta con el proceso `receive-pack` corriendo en el lado remoto.

SSH

Por ejemplo, si lanzas el comando `git push origin master` en tu proyecto y `origin` está definida como una ubicación que utiliza el protocolo SSH. Git lanzará el proceso `send-pack`, con el que establece conexión SSH con tu servidor. En el servidor remoto, a través de una llamada SSH, intentará lanzar un comando tal como:

```
$ ssh -x git@server "git-receive-pack 'simplegit-progit.git'"
005bca82a6dff817ec66f4437202690a93763949 refs/heads/master report-status \
    delete-refs side-band-64k quiet ofs-delta \
    agent=git/2:2.1.1+github-607-gfba4028 delete-refs
003e085bb3bcb608e1e84b2432f8ecbe6306e7e7 refs/heads/topic
0000
```

El comando `git-receive-pack` responde con una linea por cada una de las referencias que tenga, --en este caso, la rama `master` y su SHA-1--. La primera línea también tiene una lista de las capacidades del servidor (en este caso, `report-status`, `delete-refs`, y algunas otras, incluyendo el identificador del cliente).

Cada línea comienza con 4 caracteres, con valor en hexadecimal, indicando la longitud del resto de la línea. La primera de las líneas comienza con 005b, valor hexadecimal de 91, indicándonos que hay 91 bytes más en esa línea. La siguiente línea comienza con 003e, 62 en decimal, por lo que has de leer otros 62 bytes hasta el final de la línea. Y la última línea comienza con 0000, indicando así que la lista de referencias ha terminado.

Con esta información, el proceso `send-pack` ya puede determinar las confirmaciones de cambios (commits) no presentes en el servidor. Para cada una de las referencias que se van a actualizar, el proceso `send-pack` llama al proceso `receive-pack` con la información pertinente. Por ejemplo, si estás actualizando la rama `master` y añadiendo otra rama `experiment`, la respuesta del proceso `send-pack` será algo así como:

```
0085ca82a6dff817ec66f44342007202690a93763949  15027957951b64cf874c3557a0f3547bd83b3ff6
\
    refs/heads/master report-status
006700000000000000000000000000000000000000000 cdfdb42577e2506715f8cfeacdbabc092bf63e8d
\
    refs/heads/experiment
0000
```

Git envía una línea por cada referencia a actualizar, indicando la longitud de la línea, el viejo SHA-1, el nuevo SHA-1, y la referencia a actualizar. La primera línea indica también las capacidades disponibles en el cliente. Una clave SHA-1 con todo '0's, nos indica que no había nada anteriormente, y que, por tanto, estamos añadiendo una nueva referencia. Si estuvieras borrando una referencia existente, verías lo contrario: una clave todo '0's en el lado derecho.

A continuación, el cliente envía un archivo empaquetado con todos los objetos que faltan en el servidor. Y, por ultimo, el servidor responde con un indicador de éxito (o fracaso) de la operación:

```
000Aunpack ok
```

HTTP(S)

Este proceso es, en general, el mismo con HTTP, aunque la negociación (handshaking) es un poco diferente. La conexión se inicia con esta solicitud:

```
=> GET http://server/simplegit-progit.git/info/refs?service=git-receive-pack
001f# service=git-receive-pack
000000ab6c5f0e45abd7832bf23074a333f739977c9e8188 refs/heads/master \
    report-status delete-refs side-band-64k quiet ofs-delta \
    agent=git/2:2.1.1~vmg-bitmaps-bugaloo-608-g116744e
0000
```

Este es el final del primer intercambio cliente-servidor. El cliente, entonces, realiza otra petición, esta vez un `POST`, con los datos que proporciona `git-upload-pack`.

```
=> POST http://server/simplegit-progit.git/git-receive-pack
```

La solicitud `POST` incluye la salida de `send-pack` y el archivo empaquetado como su carga útil. Después, el servidor indica el éxito o el fracaso con su respuesta HTTP.

Recibiendo datos (downloading)

Cuando descargas datos, los procesos que se ven envueltos son `fetch-pack` (recuperar paquete) y `upload-pack` (enviar paquete). El cliente arranca un proceso `fetch-pack`, para conectar con un proceso `upload-pack` en el lado servidor y negociar con él los datos a transferir.

SSH

Si realizas la recuperación (fetch) sobre SSH, entonces `fetch-pack` ejecuta algo como:

```
$ ssh -x git@server "git-upload-pack 'simplegit-progit.git'"
```

Después de establecer conexión, `upload-pack` responderá:

```
00dfca82a6dff817ec66f44342007202690a93763949 HEADmulti_ack thin-pack \
    side-band side-band-64k ofs-delta shallow no-progress include-tag \
    multi_ack_detailed symref=HEAD:refs/heads/master \
    agent=git/2:2.1.1+github-607-gfba4028
003fca82a6dff817ec66f44342007202690a93763949 refs/heads/master
0000
```

La respuesta es muy similar a la dada por `receive-pack`, pero las capacidades que se indican son diferentes. Además, nos indica a qué apunta HEAD (`symref=HEAD:refs/heads/master`) para que el cliente pueda saber qué ha de activar (check out) en el caso de estar requiriendo un clon.

En este punto, el proceso `fetch-pack` revisa los objetos que tiene y responde indicando los objetos que necesita, enviando '*want*' (quiero) y la clave SHA-1 que necesita. Los objetos que ya tiene, los envía con '*have*' (tengo) y la correspondiente clave SHA-1. Llegando al final de la lista, escribe '*done*' (hecho), para indicar al proceso `upload-pack` que ya puede comenzar a enviar el archivo empaquetado con los datos requeridos:

```
0054want ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
0000
0009done
```

HTTP(S)

La negociación (handshake) para una operación de recuperación (fetch) requiere dos peticiones HTTP. La primera es un `GET` al mismo destino usado en el protocolo tonto:

```
=> GET $GIT_URL/info/refs?service=git-upload-pack
001e# service=git-upload-pack
000000e7ca82a6dff817ec66f44342007202690a93763949 HEADmulti_ack thin-pack \
    side-band side-band-64k ofs-delta shallow no-progress include-tag \
    multi_ack_detailed no-done symref=HEAD:refs/heads/master \
    agent=git/2:2.1.1+github-607-gfba4028
003fca82a6dff817ec66f44342007202690a93763949 refs/heads/master
0000
```

Esto es muy parecido a invocar `git-upload-pack` sobre una conexión SSH, pero el segundo intercambio es realizado como una petición separada:

```
=> POST $GIT_URL/git-upload-pack HTTP/1.0
0032want 0a53e9ddeaddad63ad106860237bbf53411d11a7
0032have 441b40d833fdfa93eb2908e52742248faf0ee993
0000
```

De nuevo, este es el mismo formato visto más arriba. La respuesta a esta petición será éxito o fallo, e incluirá el empaquetado.

Resumen

Esta sección contiene una descripción muy básica de los protocolos de transferencia. El protocolo incluye muchas otras características, como las capacidades `multi_ack` o `side-band`, pero su tratamiento está fuera del alcance de este libro. Hemos tratado de darte una panorámica de la comunicación entre cliente y servidor; si necesitas profundizar en esto, es probable que desees echar un vistazo al código fuente de Git.

Mantenimiento y recuperación de datos

De vez en cuando, es posible que necesites hacer algo de limpieza, (compactar un repositorio, adecuar un repositorio importado, recuperar trabajo perdido,...). En ese apartado vamos a ver algunos de esos escenarios.

Mantenimiento

De cuando en cuando, Git lanza automáticamente un comando llamado *'auto gc'*. La mayor parte de las veces, este comando no hace nada. Pero, cuando hay demasiados objetos sueltos, (objetos fuera de un archivo empaquetador), o demasiados archivos empaquetadores, Git lanza un comando `git gc` completo. *'gc'* corresponde a *'recogida de basura'* (garbage collect), y este comando realiza toda una serie de acciones: recoge los objetos sueltos y los agrupa en archivos empaquetadores; consolida los archivos empaquetadores pequeños en un solo gran archivo empaquetador; retira los objetos antiguos no incorporados a ninguna confirmación de cambios.

También puedes lanzar *'auto gc'* manualmente:

```
$ git gc --auto
```

Y, habitualmente, no hará nada. Ya que es necesaria la presencia de unos 7.000 objetos sueltos o más de 50 archivos empaquetadores para que Git termine lanzando realmente un comando "gc". Estos límites pueden configurarse con las opciones de configuración `gc.auto` y `gc.autopacklimit`, respectivamente.

Otra tarea realizada por `gc` es el empaquetar referencias en un solo archivo. Por ejemplo, suponiendo que tienes las siguientes ramas y etiquetas en tu repositorio:

```
$ find .git/refs -type f
.git/refs/heads/experiment
.git/refs/heads/master
.git/refs/tags/v1.0
.git/refs/tags/v1.1
```

Lanzando el comando `git gc`, dejarás de tener esos archivos en la carpeta `refs`. En aras de la eficiencia, Git los moverá a un archivo denominado `.git/packed-refs`:

```
$ cat .git/packed-refs
# pack-refs with: peeled fully-peeled
cac0cab538b970a37ea1e769cbbde608743bc96d refs/heads/experiment
ab1afef80fac8e34258ff41fc1b867c702daa24b refs/heads/master
cac0cab538b970a37ea1e769cbbde608743bc96d refs/tags/v1.0
9585191f37f7b0fb9444f35a9bf50de191beadc2 refs/tags/v1.1
^1a410efbd13591db07496601ebc7a059dd55cfe9
```

Si actualizas alguna de las referencias, Git no modificará este archivo, sino que, en cambio, escribirá uno nuevo en `refs/heads`. Para obtener la clave SHA-1 correspondiente a una determinada referencia, Git comprobará primero en la carpeta `refs` y luego en el archivo `packed-refs`. Cualquier referencia que no puedas encontrar en la carpeta `refs`, es muy posible que la encuentres en el archivo `packed-refs`.

Merece destacar que la última línea de este archivo comenzaba con `^`. Esto nos indica que la etiqueta inmediatamente anterior es una etiqueta anotada y que esa línea es la confirmación de cambios a la que apunta dicha etiqueta anotada.

Recuperación de datos

En algún momento de tu trabajo con Git, perderás por error una confirmación de cambios. Normalmente, esto suele suceder porque has forzado el borrado de una rama con trabajos no confirmados en ella, y luego te has dado cuenta de que realmente necesitabas dicha rama; o porque has reculado (hard-reset) una rama, abandonando todas sus confirmaciones de cambio, y luego te has dado cuenta que necesitabas alguna de ellas. Asumiendo que estas cosas pasan, ¿cómo podrías recuperar tus confirmaciones de cambio perdidas?

Vamos a ver un ejemplo de un retroceso a una confirmación (commit) antigua en la rama principal de tu repositorio de pruebas, y cómo podríamos recuperar las confirmaciones perdidas en este caso. Lo primero es revisar el estado de tu repositorio en ese momento:

```
$ git log --pretty=oneline
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Después, al mover la rama `master` de vuelta a la confirmación de cambios intermedia:

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
HEAD is now at 1a410ef third commit
$ git log --pretty=oneline
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Vemos que se han perdido las dos últimas confirmaciones de cambios, --no tienes ninguna rama que te permita acceder a ellas--. Necesitas localizar el SHA-1 de la última confirmación de cambios y luego añadir una rama que apunte hacia ella. El problema es cómo localizarla, --porque, ¿no te la sabrás de memoria, no?--.

El método más rápido para conseguirlo suele ser utilizar una herramienta denominada `git reflog`. Según trabajas, Git suele guardar un silencioso registro de donde está HEAD en cada momento. Cada vez que confirmas cambios o cambias de rama, el registro (reflog) es actualizado. El registro reflog se actualiza incluso cuando utilizas el comando `git update-ref`, siendo ésta otra de las razones por las que es recomendable utilizar ese comando en lugar de escribir manualmente los valores SHA-1 en los archivos de referencia, tal y como hemos visto en Referencias Git. Con el comando `git reflog` puedes revisar donde has estado en cualquier momento pasado:

```
$ git reflog
1a410ef HEAD@{0}: reset: moving to 1a410ef
ab1afef HEAD@{1}: commit: modified repo.rb a bit
484a592 HEAD@{2}: commit: added repo.rb
```

Se pueden ver las dos confirmaciones de cambios que hemos activado, pero no hay mucha más información al respecto. Para ver la misma información de manera mucho más amigable, podemos utilizar el comando `git log -g`, que nos muestra una salida normal de registro:

```
$ git log -g
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Reflog: HEAD@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:22:37 2009 -0700

        third commit

commit ab1afef80fac8e34258ff41fc1b867c702daa24b
Reflog: HEAD@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700

       modified repo.rb a bit
```

Parece que la confirmación de cambios perdida es esta última, así que puedes recuperarla creando una nueva rama apuntando a ella. Por ejemplo, puedes iniciar una rama llamada `recover-branch` con dicha confirmación de cambios (ab1afef):

```
$ git branch recover-branch ab1afef
$ git log --pretty=oneline recover-branch
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

¡Bravo!, acabas de añadir una rama denominada `recover-branch` al punto donde estaba originalmente tu rama `master`; permitiendo así recuperar el acceso a las dos primeras confirmaciones de cambios.

A continuación, supongamos que la pérdida se ha producido por alguna razón que no se refleja en el registro (reflog) --puedes simularlo borrando la rama `recover-branch` y borrando asimismo el registro--. Con eso, perdemos completamente el acceso a las dos primeras confirmaciones de cambio:

```
$ git branch -D recover-branch
$ rm -Rf .git/logs/
```

La información de registro (reflog) se guarda en la carpeta `.git/logs/`; por lo que, borrándola, nos quedamos efectivamente sin registro. ¿Cómo podríamos ahora recuperar esas confirmaciones de cambio? Una forma es utilizando el comando de chequeo de integridad de la base de datos: `git fsck`. Si lo lanzas con la opción `--full`, te mostrará todos los objetos sin referencias a ningún otro objeto:

```
$ git fsck --full
Checking object directories: 100% (256/256), done.
Checking objects: 100% (18/18), done.
dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4
dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b
dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9
dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293
```

En este caso, puedes ver la confirmación de cambios perdida después del texto '*dangling commit*'. Y la puedes recuperar del mismo modo, añadiendo una rama que apunte a esa clave SHA-1.

Borrando objetos

Git tiene grandes cosas, pero el hecho de que un `git clone` siempre descargue la historia completa del proyecto (incluyendo todas y cada una de las versiones de todos y cada uno de los archivos) puede causar problemas. Todo suele ir bien si el contenido es únicamente código fuente, ya que Git está tremendamente optimizado para comprimir eficientemente ese tipo de datos. Pero, si alguien, en cualquier momento de tu proyecto, ha añadido un solo archivo enorme, a partir de ese momento, todos los clones, siempre, se verán obligados a copiar ese enorme archivo, incluso si ya ha sido borrado del proyecto en la siguiente confirmación de cambios realizada inmediatamente tras la que lo añadió. Porque en algún momento formó parte del proyecto, siempre permanecerá ahí.

Esto suele dar bastantes problemas cuando estás convirtiendo repositorios de Subversion o de Perforce a Git. En esos sistemas, uno no se suele descargar la historia completa y, por tanto, los archivos enormes no tienen mayores consecuencias. Si, tras una importación de otro sistema, o por otras razones, descubres que tu repositorio es mucho mayor de lo que debería ser, es momento de buscar y borrar objetos enormes en él.

Una advertencia importante: estas técnicas son destructivas y alteran el historial de confirmaciones de cambio. Se basan en reescribir todos los objetos confirmados desde el árbol más reciente modificado para borrar la referencia a un archivo enorme. No tendrás problemas si lo haces inmediatamente después de una importación, o justo antes de que alguien haya comenzado a trabajar con la confirmación de cambios modificada --pero si no es el caso, tendrás que avisar a todas las personas que hayan contribuido para que reorganicen su trabajo en base a tus nuevas confirmaciones de cambio--.

Para probarlo por ti mismo, puedes añadir un archivo enorme a tu repositorio de pruebas y retirarlo en la siguiente confirmación de cambios; así podrás practicar la búsqueda y borrado permanente del repositorio. Para empezar, añade un objeto enorme a tu historial:

```
$ curl https://www.kernel.org/pub/software/scm/git/git-2.1.0.tar.gz > git.tgz
$ git add git.tgz
$ git commit -m 'add git tarball'
[master 7b30847] add git tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 git.tgz
```

!Ouch!, --no querías añadir un archivo tan grande a tu proyecto--. Mejor si lo quitas:

```
$ git rm git.tgz
rm 'git.tgz'
$ git commit -m 'oops - removed large tarball'
[master dadf725] oops - removed large tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 git.tgz
```

Ahora, puedes limpiar con `gc` tu base de datos y comprobar cuánto espacio estás ocupando:

```
$ git gc
Counting objects: 17, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (17/17), done.
Total 17 (delta 1), reused 10 (delta 0)
```

Puedes utilizar el comando `count-objects` para revisar rápidamente el espacio utilizado:

```
$ git count-objects -v
count: 7
size: 32
in-pack: 17
packs: 1
size-pack: 4868
prune-packable: 0
garbage: 0
size-garbage: 0
```

El valor de `size-pack` nos da el tamaño de tus archivos empaquetadores, en kilobytes, y, por lo que se ve, estás usando casi 5MB. Antes de la última confirmación de cambios, estabas usando algo así como 2KB --resulta claro que esa última confirmación de cambios no ha borrado el archivo enorme del historial--. A partir de este momento, cada vez que alguien haga un clon de este repositorio, se verá obligado a copiar 5MB para un proyecto tan simple, y todo porque tu añadiste accidentalmente un archivo enorme en algún momento. Deshagámonos de él.

Lo primero es localizarlo. En este caso, ya sabes de antemano cual es. Pero suponiendo que no lo supieras, ¿cómo podrías identificar el archivo o archivos que están ocupando tanto espacio?. Tras lanzar el comando `git gc`, todos los objetos estarán guardados en un archivo empaquetador; puedes identificar los objetos enormes en su interior, utilizando otro comando de fontanería denominado `git verify-pack` y ordenando su salida por su tercera columna, la que nos informa de los tamaños de cada objeto. Puedes también redirigir su salida a través del comando `tail`, porque realmente solo nos interesan las últimas líneas, las correspondientes a los archivos más grandes.

```
$ git verify-pack -v .git/objects/pack/pack-29…69.idx \
  | sort -k 3 -n \
  | tail -3
dadf7258d699da2c8d89b09ef6670edb7d5f91b4 commit 229 159 12
033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 blob   22044 5792 4977696
82c99a3e86bb1267b236a4b6eff7868d97489af1 blob   4975916 4976258 1438
```

El archivo enorme es el último: 5MB. Para concretar cual es el archivo, puedes utilizar el comando `rev-list` que ya vimos brevemente en Obligando a utilizar un formato específico en el mensaje de commit. Con la opción `--objects`, obtendrás la lista de todas las SHA-1s de todas las confirmaciones de cambio, junto a las SHA-1s de los objetos binarios y las ubicaciones (paths) de cada uno de ellos. Puedes usar esta información para localizar el nombre del objeto binario:

```
$ git rev-list --objects --all | grep 82c99a3
82c99a3e86bb1267b236a4b6eff7868d97489af1 git.tgz
```

Una vez tengas ese dato, lo puedes utilizar para borrar ese archivo en todos los árboles pasados. Es sencillo revisar cuales son las confirmaciones de cambios donde interviene ese archivo:

```
$ git log --oneline --branches -- git.tgz
dadf725 oops - removed large tarball
7b30847 add git tarball
```

Para borrar realmente ese archivo de tu historial Git, has de reescribir todas las confirmaciones de cambio desde `7b30847`. Y, para ello, puedes emplear el comando `filter-branch` que se vió en Reescribiendo la Historia:

```
$ git filter-branch --index-filter \
  'git rm --cached --ignore-unmatch git.tgz' -- 7b30847^..
Rewrite 7b30847d080183a1ab7d18fb202473b3096e9f34 (1/2)rm 'git.tgz'
Rewrite dadf7258d699da2c8d89b09ef6670edb7d5f91b4 (2/2)
Ref 'refs/heads/master' was rewritten
```

La opción `--index-filter` es similar a la `--tree-filter` vista en Reescribiendo la Historia, pero, en lugar de modificar archivos activados (checked out) en el disco, se modifica el área de preparación (staging area) o índice. En lugar de borrar un archivo concreto con una orden tal como `rm archivo`, has de borrarlo con `git rm --cached` (es decir, tienes que borrarlo del índice, en lugar de del disco). Con eso conseguimos aumentar la velocidad, ya que el proceso es mucho más rápido, porque Git no ha de activar cada revisión a disco antes de procesar tu filtro. Aunque también puedes hacer lo mismo con la opción`--tree-filter`, si así lo prefieres. La opción `--ignore-unmatch` indica a `git rm` que evite lanzar errores en caso de no encontrar el patrón que le has ordenado buscar. Y por último, se indica a `filter-branch` que reescriba la historia a partir de la confirmación de cambios `7b30847`, porque ya conocemos que es a partir de ahí donde comenzaba el problema. De otro modo, el comando comenzaría desde el principio, asumiendo un proceso innecesariamente más largo.

Tras esto, el historial ya no contiene ninguna referencia a ese archivo. Pero, sin embargo, quedan referencias en el registro (reflog) y en el nuevo conjunto de referencias en `.git/refs/original` que Git ha añadido al procesar `filter-branch`, por lo que has de borrar también éstas y reempaquetar la base de datos. Antes de reempaquetar, asegúrate de acabar completamente con cualquier elemento que apunte a las viejas confirmaciones de cambios:

```
$ rm -Rf .git/refs/original
$ rm -Rf .git/logs/
$ git gc
Counting objects: 15, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (11/11), done.
Writing objects: 100% (15/15), done.
Total 15 (delta 1), reused 12 (delta 0)
```

Y ahora, vamos a ver cuanto espacio hemos ahorrado.

```
$ git count-objects -v
count: 11
size: 4904
in-pack: 15
packs: 1
size-pack: 8
prune-packable: 0
garbage: 0
size-garbage: 0
```

El tamaño del repositorio ahora es de 8KB, mucho mejor que los 5MB anteriores. Por el valor de "size", puedes ver que el objeto enorme sigue estando entre tus objetos sueltos, es decir, no hemos acabado completamente con él; pero lo importante es que ya no será considerado al transferir o clonar el proyecto. Si realmente quieres acabar del todo, puedes lanzar la orden `git prune` con la opción `--expire` para retirar incluso ese archivo suelto.

```
$ git prune --expire now
$ git count-objects -v
count: 0
size: 0
in-pack: 15
packs: 1
size-pack: 8
prune-packable: 0
garbage: 0
size-garbage: 0
```

Variables de entorno

Git siempre se ejecuta dentro de un shell `bash`, y utiliza una serie de variables de entorno de shell para determinar cómo comportarse. En ocasiones, es muy útil saber cuáles son, y cómo pueden ser utilizadas para hacer que Git se comporte de la manera que deseas. Esta no es una lista exhaustiva de todas las variables de entorno a las que Git presta atención, pero cubriremos las más útiles.

Comportamiento Global

Como programa de computadora que es, el comportamiento general de Git depende de variables de entorno.

`GIT_EXEC_PATH` determina el sitio donde Git busca sus subprogramas (como `git-commit`, `git-diff`, y otros). Puedes comprobar la configuración actual ejecutando `git --exec-path`.

`HOME` por regla general no se considera personalizable (demasiadas cosas dependen de él), pero es el sitio donde Git busca el archivo de configuración global. Si deseas una instalación de Git realmente portable, completa con configuración global, puedes sobreescribir `HOME` en el perfil de shell portable de Git.

`PREFIX` es parecido, pero para la configuración del sistema. Git busca este archivo en `$PREFIX/etc/gitconfig`.

`GIT_CONFIG_NOSYSTEM`, si está establecida, deshabilita el uso del archivo de configuración del sistema. Esto es útil si la configuración de tu sistema está interfiriendo con tus comandos, pero no tienes acceso para modificarlo o eliminarlo.

`GIT_PAGER` controla el programa usado para mostrar la salida de varias páginas en la línea de comandos. Si no está establecida, será usado `PAGER` por defecto.

`GIT_EDITOR` es el editor que Git lanzará cuando el usuario necesite editar un texto (un mensaje para confirmación de cambio, por ejemplo). Si no está establecida, será usado `EDITOR`.

Ubicaciones del Repositorio

Git utiliza varias variables de entorno para determinar la forma en que interactúa con el repositorio actual.

`GIT_DIR` es la ubicación del directorio `.git`. Si no se especifica, Git subirá por el árbol de directorios hasta llegar a `~` o `/`, en busca de un directorio `.git` en cada directorio visitado.

`GIT_CEILING_DIRECTORIES` controla el comportamiento de búsqueda del directorio `.git`. Si accedes a directorios que son lentos de cargar (tales como aquellos en un dispositivo de cinta magnética, o a través de una conexión de red lenta), es posible que desees que Git se detenga antes de lo establecido, especialmente si Git se invoca durante la construcción de tu indicador de shell.

`GIT_WORK_TREE` es la ubicación de la raíz del directorio de trabajo para un repositorio con contenido. Si no se especifica, se utiliza el directorio padre de `$GIT_DIR`.

`GIT_INDEX_FILE` es la ruta de acceso al archivo index (solo repositorios con contenido).

`GIT_OBJECT_DIRECTORY` puede utilizarse para especificar la ubicación del directorio que normalmente reside en `.git/objects`.

`GIT_ALTERNATE_OBJECT_DIRECTORIES` es una lista separada por dos puntos (formateada como `/dir/one:/dir/two:…`) que indica a Git dónde buscar objetos si no están en `GIT_OBJECT_DIRECTORY`. Si tienes muchos proyectos con archivos grandes que tienen exactamente el mismo contenido, puedes utilizar esto para evitar almacenar demasiadas copias de ellos.

Especificaciones de Rutas de Acceso (pathspecs)

Un '*pathspec*' se refiere a la forma de especificar rutas de acceso a las cosas en Git, incluyendo el uso de comodines. Se utilizan en el archivo `.gitignore`, pero también en la línea de comandos (`git add *.c`).

`GIT_GLOB_PATHSPECS` y `GIT_NOGLOB_PATHSPECS` controlan el comportamiento por defecto de los comodines en las especificaciones de rutas de acceso. Si `GIT_GLOB_PATHSPECS` está establecida a 1, los caracteres de comodín actúan como comodines (lo cual es la situación por defecto); si `GIT_NOGLOB_PATHSPECS` está establecida a 1, los caracteres de comodín sólo coinciden consigo mismos, lo que significa que algo como `\*.c` solo coincidiría con un archivo *llamado* `*.c`, en lugar de cualquier archivo cuyo nombre acabe en `.c`. Puedes sobreescribir esto para casos individuales iniciando la especificación de ruta de acceso con `:(glob)` o `:(literal)`, como en `:(glob)*.c`.

`GIT_LITERAL_PATHSPECS` deshabilita los dos comportamientos anteriores; los caracteres de comodín no funcionarán, y la sobreescritura de prefijos estará deshabilitada también.

`GIT_ICASE_PATHSPECS` establece todas las especificaciones de rutas de acceso para que funcionen de forma que no se diferencie entre mayúsculas y minúsculas (case-insensitive).

Confirmaciones (Committing)

La creación final de un objeto de confirmación de cambios en Git se realiza generalmente con `git-commit-tree`, que utiliza estas variables de entorno como su fuente primera de información, utilizando los valores de la configuración sólo si éstas no están presentes.

`GIT_AUTHOR_NAME` es el nombre completo en el campo '*author*'.

`GIT_AUTHOR_EMAIL` es el email para el campo '*author*'.

`GIT_AUTHOR_DATE` es la fecha y hora utilizada para el campo '*author*'.

`GIT_COMMITTER_NAME` establece el nombre completo para el campo '*committer*'.

`GIT_COMMITTER_EMAIL` es la dirección de email para el campo '*committer*'.

`GIT_COMMITTER_DATE` se utiliza para la fecha y hora en el campo '*committer*'.

`EMAIL` es la dirección de email utilizada en caso de que el valor de configuración `user.email` no esté establecido. Si *éste* no está configurado, Git utilizará los nombres de usuario y equipo.

Red

Git utiliza la biblioteca `curl` para realizar operaciones de red sobre HTTP, así que `GIT_CURL_VERBOSE` le indica a Git que emita todos los mensajes generados por esta biblioteca. Esto es similar a hacer `curl -v` en la línea de comandos.

`GIT_SSL_NO_VERIFY` indica a Git que no verifique los certificados SSL. Esto a veces puede ser necesario si estás utilizando un certificado autofirmado para servir un repositorio Git a través de HTTPS, o estás montando un servidor Git, pero todavía no has instalado un certificado completo.

Si la velocidad de datos de una operación de HTTP es menor de `GIT_HTTP_LOW_SPEED_LIMIT` bytes por segundo durante más de `GIT_HTTP_LOW_SPEED_TIME` segundos, Git abortará esa operación. Estos valores sobreescriben los valores de configuración `http.lowSpeedLimit` y `http.lowSpeedTime`.

`GIT_HTTP_USER_AGENT` establece el texto de agente de usuario utilizado por Git cuando se comunica sobre HTTP. El valor predeterminado es similar a `git/2.0.0`.

Diferencias y Fusiones

`GIT_DIFF_OPTS` es un nombre poco apropiado. Los únicos valores válidos son `-u<n>` o `--unified=<n>`, que controla el número de líneas de contexto mostradas en un comando `git diff`.

`GIT_EXTERNAL_DIFF` se utiliza como para sobreescribir el valor de configuración `diff.external`. Si está establecido, Git invocará este programa cuando se ejecute `git diff`.

`GIT_DIFF_PATH_COUNTER` y `GIT_DIFF_PATH_TOTAL` son útiles para el programa especificado en `GIT_EXTERNAL_DIFF` o `diff.external`. El primero representa el archivo de la serie (empezando en 1) que está siendo comparado, y el último es el número total de archivos en el lote.

`GIT_MERGE_VERBOSITY` controla la salida de la estrategia de fusión recursiva. Los valores permitidos son los siguientes:

- 0 no muestra nada, excepto un simple mensaje de error, posiblemente.
- 1 muestra sólo los conflictos.
- 2 también muestra los cambios de los archivos.
- 3 muestra cuando los archivos son ignorados porque no han sido modificados.
- 4 muestra todas las rutas de acceso a medida que son procesadas.
- 5 y muestra información de depuración detallada.

El valor predeterminado es 2.

Depuración

¿Quieres saber lo que hace *realmente* Git? Git tiene un conjunto bastante completo de trazas incorporadas, y todo lo que hay que hacer es activarlas. Los posibles valores de estas variables son los siguientes:

- '*true*', '*1*', o '*2*' — la categoría de la traza se escribe a stderr — .

- Una ruta absuluta iniciada con `/` — la salida de la traza será escrita en ese archivo — .

`GIT_TRACE` controla las trazas generales, que no se ajustan a una categoría específica. Esto incluye la expansión de alias, y la delegación a otros subprogramas.

```
$ GIT_TRACE=true git lga
20:12:49.877982 git.c:554               trace: exec: 'git-lga'
20:12:49.878369 run-command.c:341       trace: run_command: 'git-lga'
20:12:49.879529 git.c:282               trace: alias expansion: lga => 'log' '--graph'
'--pretty=oneline' '--abbrev-commit' '--decorate' '--all'
20:12:49.879885 git.c:349               trace: built-in: git 'log' '--graph' '--
pretty=oneline' '--abbrev-commit' '--decorate' '--all'
20:12:49.899217 run-command.c:341       trace: run_command: 'less'
20:12:49.899675 run-command.c:192       trace: exec: 'less'
```

`GIT_TRACE_PACK_ACCESS` controla el trazado de acceso a empaquetados (packfile). El primer campo es el archivo empaquetado que está siendo accedido, el segundo es el desplazamiento (offset) dentro del archivo:

```
$ GIT_TRACE_PACK_ACCESS=true git status
20:10:12.081397 sha1_file.c:2088        .git/objects/pack/pack-c3fa...291e.pack 12
20:10:12.081886 sha1_file.c:2088        .git/objects/pack/pack-c3fa...291e.pack 34662
20:10:12.082115 sha1_file.c:2088        .git/objects/pack/pack-c3fa...291e.pack 35175
# […]
20:10:12.087398 sha1_file.c:2088        .git/objects/pack/pack-e80e...e3d2.pack
56914983
20:10:12.087419 sha1_file.c:2088        .git/objects/pack/pack-e80e...e3d2.pack
14303666
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

`GIT_TRACE_PACKET` habilita el trazado a nivel de paquete para las operaciones de red.

```
$ GIT_TRACE_PACKET=true git ls-remote origin
20:15:14.867043 pkt-line.c:46           packet:          git< # service=git-upload-
pack
20:15:14.867071 pkt-line.c:46           packet:          git< 0000
20:15:14.867079 pkt-line.c:46           packet:          git<
97b8860c071898d9e162678ea1035a8ced2f8b1f HEAD\0multi_ack thin-pack side-band side-
band-64k ofs-delta shallow no-progress include-tag multi_ack_detailed no-done
symref=HEAD:refs/heads/master agent=git/2.0.4
20:15:14.867088 pkt-line.c:46           packet:          git<
0f20ae29889d61f2e93ae00fd34f1cdb53285702 refs/heads/ab/add-interactive-show-diff-func-
name
20:15:14.867094 pkt-line.c:46           packet:          git<
36dc827bc9d17f80ed4f326de21247a5d1341fbc refs/heads/ah/doc-gitk-config
# […]
```

`GIT_TRACE_PERFORMANCE` controla el registro de datos de rendimiento. La salida muestra cuánto dura cada invocación particular de git.

```
$ GIT_TRACE_PERFORMANCE=true git gc
20:18:19.499676 trace.c:414             performance: 0.374835000 s: git command: 'git'
'pack-refs' '--all' '--prune'
20:18:19.845585 trace.c:414             performance: 0.343020000 s: git command: 'git'
'reflog' 'expire' '--all'
Counting objects: 170994, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (43413/43413), done.
Writing objects: 100% (170994/170994), done.
Total 170994 (delta 126176), reused 170524 (delta 125706)
20:18:23.567927 trace.c:414             performance: 3.715349000 s: git command: 'git'
'pack-objects' '--keep-true-parents' '--honor-pack-keep' '--non-empty' '--all' '--
reflog' '--unpack-unreachable=2.weeks.ago' '--local' '--delta-base-offset'
'.git/objects/pack/.tmp-49190-pack'
20:18:23.584728 trace.c:414             performance: 0.000910000 s: git command: 'git'
'prune-packed'
20:18:23.605218 trace.c:414             performance: 0.017972000 s: git command: 'git'
'update-server-info'
20:18:23.606342 trace.c:414             performance: 3.756312000 s: git command: 'git'
'repack' '-d' '-l' '-A' '--unpack-unreachable=2.weeks.ago'
Checking connectivity: 170994, done.
20:18:25.225424 trace.c:414             performance: 1.616423000 s: git command: 'git'
'prune' '--expire' '2.weeks.ago'
20:18:25.232403 trace.c:414             performance: 0.001051000 s: git command: 'git'
'rerere' 'gc'
20:18:25.233159 trace.c:414             performance: 6.112217000 s: git command: 'git'
'gc'
```

`GIT_TRACE_SETUP` muestra información que Git va descubriendo sobre el repositorio y el entorno con el que interactúa.

```
$ GIT_TRACE_SETUP=true git status
20:19:47.086765 trace.c:315             setup: git_dir: .git
20:19:47.087184 trace.c:316             setup: worktree: /Users/ben/src/git
20:19:47.087191 trace.c:317             setup: cwd: /Users/ben/src/git
20:19:47.087194 trace.c:318             setup: prefix: (null)
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

Varios

GIT_SSH, si se especifica, es un programa que se invoca en lugar de ssh cuando Git intenta conectar a un equipo SSH. Se invoca como $GIT_SSH [username@]host [-p <port>] <command>. Observa que esta no es la forma más fácil de personalizar como ssh es invocado; no soportará parámetros extra en la línea de comandos, de modo que tendrías que escribir un script envoltorio y hacer que GIT_SSH apunte a él. Es probablemente más fácil usar simplemente el archivo ~/.ssh/config para esto.

GIT_ASKPASS es una sobreescritura para el valor de configuración core.askpass. Este es el programa invocado cada vez que Git necesita pedir al usuario las credenciales, que puede mostrar un indicador similar a la línea de comandos, y debería devolver la respuesta en stdout. (Consulta Almacenamiento de credenciales para más detalles acerca de este subsistema.)

GIT_NAMESPACE controla el acceso a las referencias de un espacio de nombres, y es equivalente al indicador --namespace. Esto es principalmente útil en el lado servidor, donde puedes querer almacenar múltiples bifurcaciones (forks) de un único repositorio en un repositorio, solo manteniendo las referencias separadas.

GIT_FLUSH puede usarse para forzar a Git a usar Entrada/Salida sin buffer cuando escribe de forma incremental a la salida estándar (stdout) . Un valor de 1 hace que Git actualice más a menudo, un valor de 0 hace que toda la salida sea con buffer. El valor por defecto (si esta variable no está definida) es elegir un esquema de buffer adecuado en función de la actividad y el modo de salida.

GIT_REFLOG_ACTION te permite especificar el texto descriptivo escrito en el registro de referencias. Aquí tienes un ejemplo:

```
$ GIT_REFLOG_ACTION="my action" git commit --allow-empty -m 'my message'
[master 9e3d55a] my message
$ git reflog -1
9e3d55a HEAD@{0}: my action: my message
```

Recapitulación

A estas alturas deberías tener una idea bastante clara de como trabaja Git entre bastidores y, hasta cierto punto, sobre cómo está implementado. En este capítulo se han visto unos cuantos comandos "de fontanería" -comandos de menor nivel y más simples que los "de porcelana" que hemos estado viendo en el resto del libro. Entendiendo cómo trabaja Git a bajo nivel, es más sencillo comprender por qué hace lo que hace, a la par que facilita la escritura de tus propias herramientas y scripts

auxiliares para implementar flujos de trabajo tal y como necesites.

Git, en su calidad de sistema de archivos de contenido localizable, es una herramienta muy poderosa que puedes usar fácilmente más que sólo un sitema de control de versiones. Esperamos que uses este nuevo conocimiento profundo de las entrañas de Git para implementar tus propias aplicaciones y para que te encuentres más cómodo usando Git de forma avanzada.

Git en otros entornos

Si has leído hasta aquí todo el libro, seguro que has aprendido un montón de cosas sobre el uso de Git con la línea de comandos. Se puede trabajar con archivos locales, conectar nuestro repositorio con otros repositorios en la red y realizar nuestro trabajo eficientemente con ellos. Aunque las opciones no terminan ahí, Git se utiliza con parte de un ecosistema mayor y un terminal no siempre es la mejor forma de trabajar. Vamos a ver otros tipos de entornos en los que Git resulta muy útil y cómo otras aplicaciones (incluidas las tuyas) pueden trabajar conjuntamente con Git.

Interfaces gráficas

El entorno nativo de Git es la línea de comandos. Sólo desde la línea de comandos se encuentra disponible todo el poder de Git. El texto plano no siempre es la mejor opción para todas las tareas y en ocasiones se necesita una representación visual además, algunos usuarios se siente más cómodo con una interfaz de apuntar y pulsar.

Conviene advertir que los diferentes interfaces están adaptados para diferentes flujos de trabajo. Algunos clientes sólo disponen de un subconjunto adecuadamente seleccionado de toda la funcionalidad de Git para poder atender una forma concreta de trabajar que los autores consideran eficiente. Bajo esta perspectiva, ninguna de estas herramientas puede considerase “mejor” que otras, simplemente se ajusta mejor a un objetivo prefijado. Además, no hay nada en estos clientes gráficos que no se encuentre ya en el cliente en línea de comandos y, por tanto, la línea de comandos es el modo con el que se consigue la máxima potencia y control cuando se trabaja con repositorios.

`gitk` y `git-gui`

Cuando se instala Git, también se instalan sus herramientas gráficas: `gitk` y `git-gui`.

`gitk` es un visor gráfico del histórico. Hay que considerarlo como una interfaz gráfica mejorada sobre `git log` y `git grep`. Es la herramienta que hay que utilizar cuando se quiere encontrar algo que sucedió en el pasado o visualizar el histórico de un proyecto.

Gitk es muy fácil de invocar desde la línea de comandos. Simplemente, hay que moverse con `cd` hasta un repositorio de Git y teclear:

```
$ gitk [git log options]
```

Gitk admite muchas opciones desde la línea de comandos, la mayoría de las cuales se pasan desde la acción `git log` subyacente. Probablemente una de las más útiles sea la opción `--all` que indica a gitk que muestre todos los commit accesibles desde *cualquier* referencia, no sólo desde el HEAD. La interfaz de gitk tiene este aspecto:

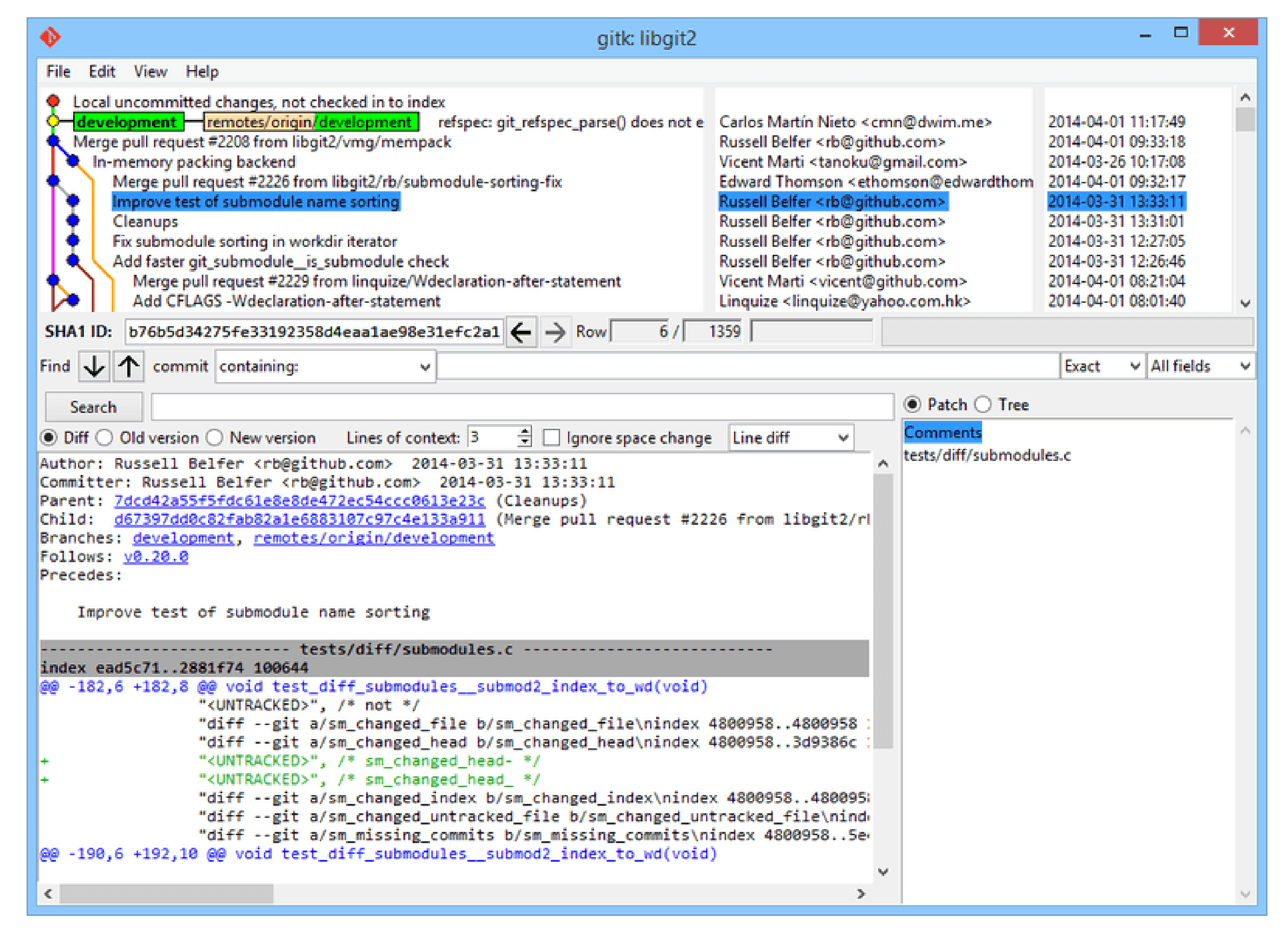


Figure 153. El visor de históricos `gitk`.

En la parte superior hay algo que se parece un poco a la salida de `git log --graph`, donde cada punto representa un commit, las líneas representan relaciones entre padres e hijos y las referencias aparecen como cajas coloreadas. El punto amarillo representa una CABEZA o HEAD y el punto rojo a cambios que han sido aceptado con commit. En la parte inferior se encuentra un commit seleccionado con los comentarios y el parche a la izquierda, junto con una vista resumen a la derecha. En medio hay una serie de controles que se usan para buscar en el histórico.

Por su parte, `git-gui` es principalmente una herramienta para elaborar commits. Resulta igual de sencilla de invocar desde la línea de comandos:

```
$ git gui
```

La herramienta tiene este aspecto:

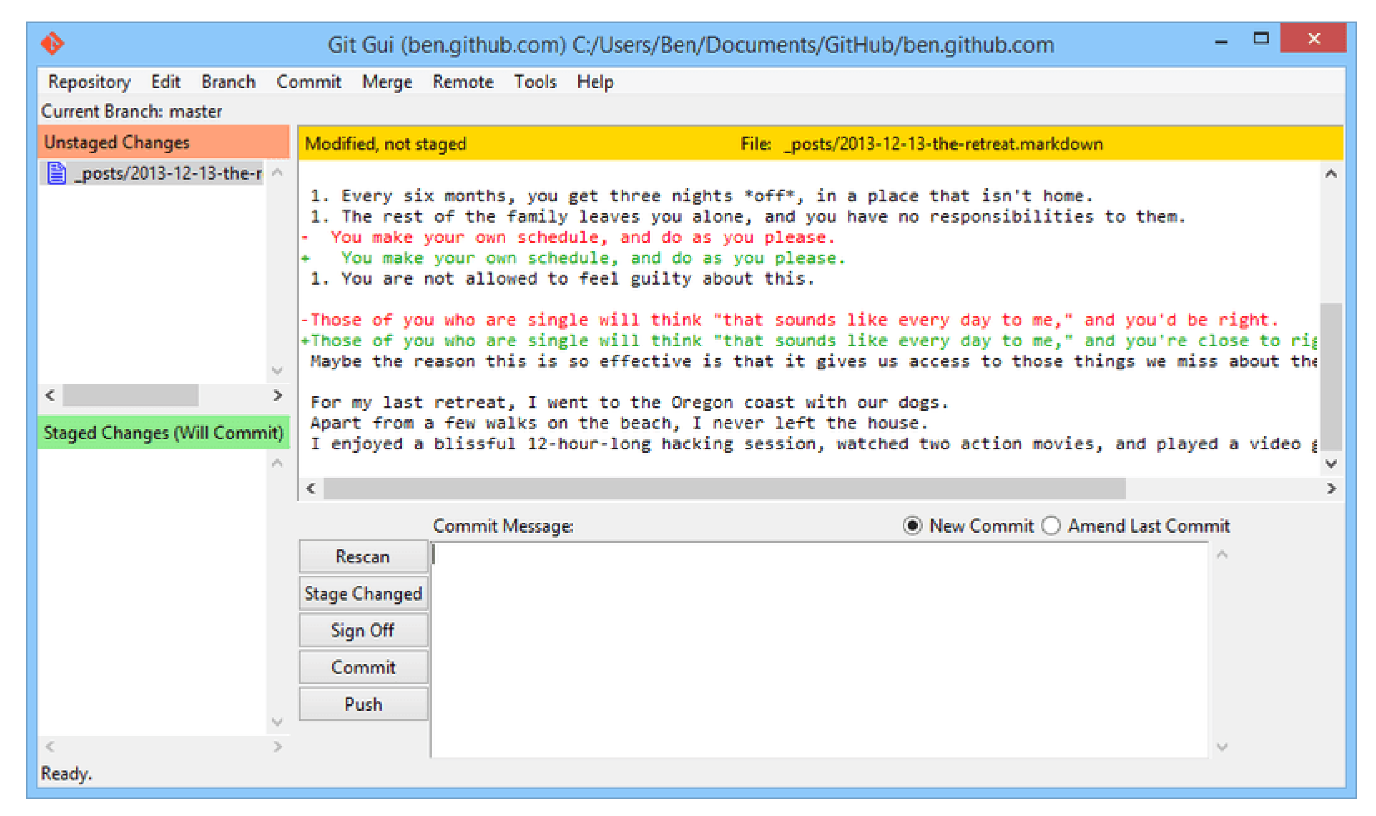


Figure 154. La herramienta `git-gui`.

A la izquierda está el índice con los cambios no preparados al principio y los preparados al final. Es posible mover archivos completos entre los dos estados haciendo clic en sus respectivos iconos o seleccionar un archivo para visualizarlo haciendo clic en su nombre.

A la derecha, en la parte superior, se encuentra la vista de diferencias que muestra los cambios producidos en el archivo seleccionado. Se pueden preparar bloques o líneas individualmente haciendo clic con el botón derecho sobre esta zona.

En la parte inferior se encuentra el área de mensajes y acciones. Escribiendo un mensaje en la caja de texto y haciendo clic en el botón “Commit`` se hace algo lo mismo que con `git commit`. Es posible rectificar el último commit eligiendo ``Amend`` en el botón de opción, con lo que se actualizará la zona de cambios preparados o ``Staged Changes`` con el contenido del último commit. Así es posible poner un cambio como preprepado o no preparado, modificar el mensaje de un commit y, a continuación, hacer clic en "Commit” para sustituir un commit anterior por uno nuevo.

`gitk` y `git-gui` son ejemplos de herramientas orientadas a tareas. Cada una de ellas está adaptada para un propósito específico (visualización del histórico y creación de commits, respectivamente), omitiendo aquellas características que no son necesarias para su tarea.

GitHub para Mac y Windows

GitHub ha creado dos clientes Git: uno para Windows y otro para Mac. Estos clientes son un claro ejemplo de herramientas orientadas a flujo de trabajo, en vez de proporcionar *toda* la funcionalidad de Git, se centran en conjunto conservador de las funcionalidades más utilizadas que colaboran eficazmente. Tienen este aspecto:

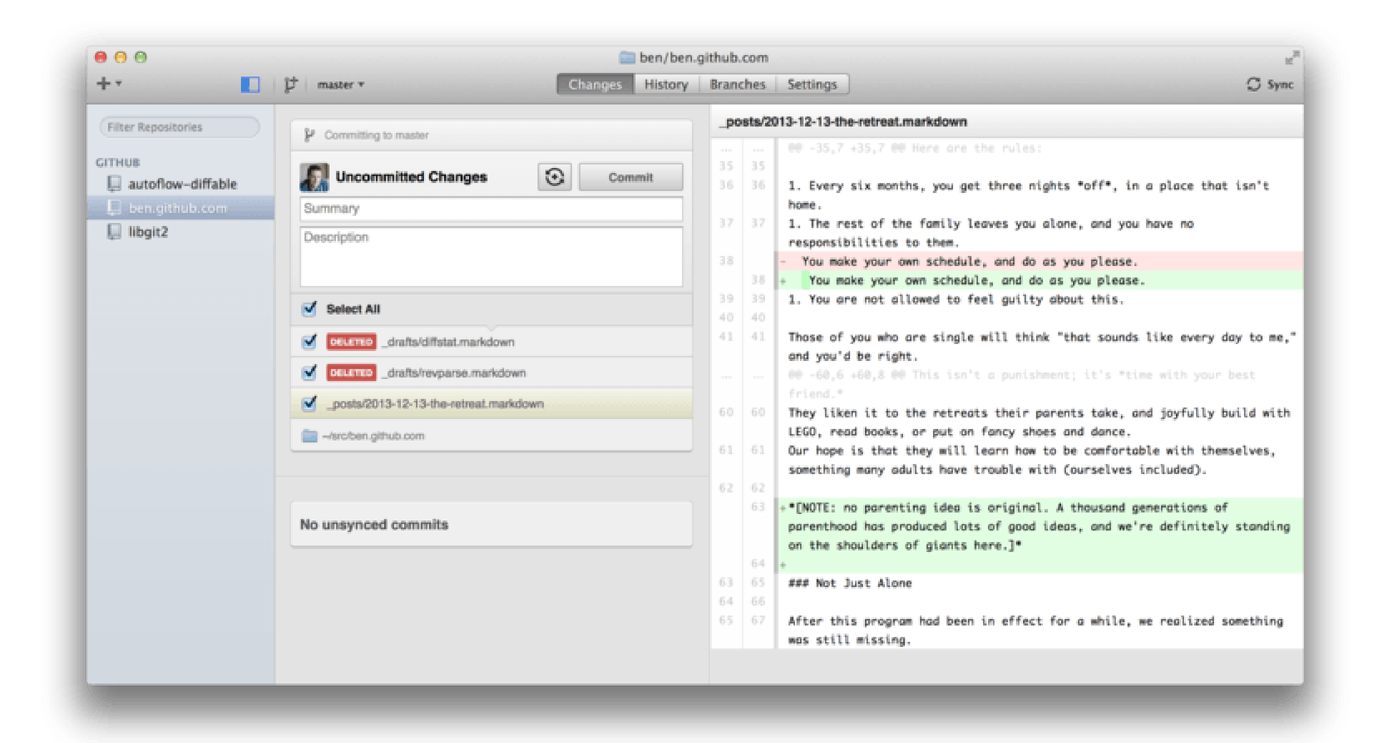


Figure 155. GitHub para Mac.

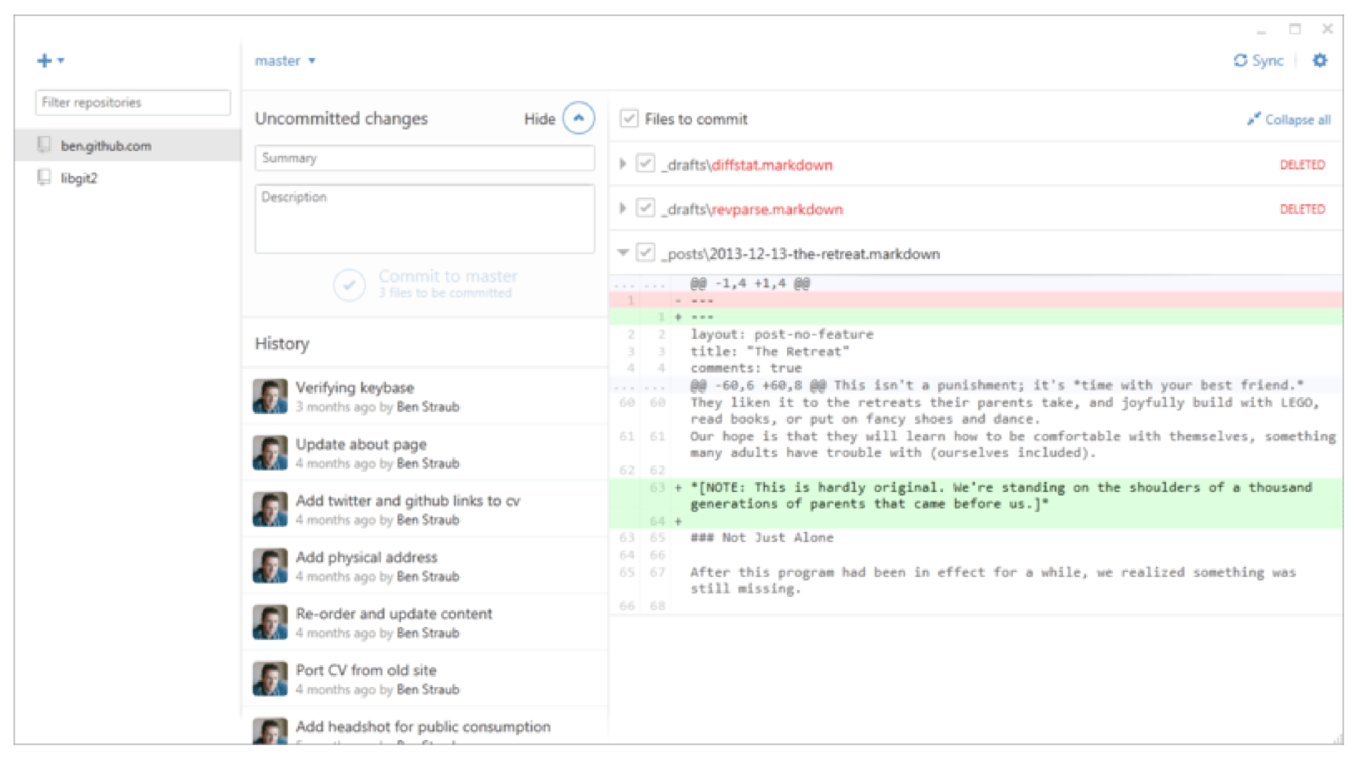


Figure 156. GitHub para Windows.

Se diseñaron para parecer y funcionar de una forma muy similar, por tanto, en este capítulo las trataremos como un mismo producto. Aquí no encontrarás un informe detallado de estas herramientas (para eso tienen su propia documentación), aunque sí hay un recorrido por la vista de “cambios” (que es donde pasaremos la mayor parte del tiempo).

- A la izquierda se encuentra la lista de repositorios que el cliente tiene en seguimiento. Se pueden añadir un repositorio (bien por clonación o agregándolo localmente) pulsado en el icono “+” que está en la parte superior de este área.

- En el centro está el área de entrada de commit, en la que puedes introducir el mensaje del commit y seleccionar los archivos que se tienen que incluir. (En Windows, se visualiza el histórico de commit directamente debajo mientras que en Mac, lo hace en una pestaña separada.)
- En la parte derecha está el visor de diferencias que muestra los cambios producidos en el directorio de trabajo o los cambios que se incluyeron en el commit seleccionado.
- Por último, se debe tener en cuenta que el botón "Sync", en la parte superior derecha, es el principal método de interactuar con la red.

NOTE	Para usar estas herramientas no es necesario tener una cuenta en GitHub. Aunque se diseñaron como punto fuerte del servicio GitHub y del flujo de trabajo recomendado, realmente funcionan con cualquier repositorio y pueden realizar las operaciones de red con cualquier servidor Git.

Instalación

GitHub para Windows se puede descargar de https://windows.github.com y GitHub para Mac de https://mac.github.com. Cuando la aplicación se ejecuta por primera vez, realiza un recorrido por toda la configuración inicial, es decir, se configura el nombre y la dirección de correo electrónico, así como realiza las configuraciones normales para muchas opciones de configuración habituales como la caché de credenciales o el comportamiento del salto de línea (CRLF).

Ambos están "siempre actualizados" - las actualizaciones se descargan e instalan en segundo plano mientras la aplicación está abierta. Incluyen un práctica versión completa de Git que significa que muy probablemente no tengas que preocuparte más de actualizarlo manualmente nunca más. El cliente de Windows incluye un acceso directo para lanzar un Powershell con Posh-git, que veremos más adelante en este capítulo.

El siguiente paso consiste en indicarle a la herramienta algún repositoria para que comience comience a trabajar. El cliente te muestra una lista con los repositorios a los que tienes acceso en GitHub para que puedas clonarlos en un solo paso. Si también dispones de repositorios locales, simplemente arrastra los directorios desde Finder o desde el Explorador de Windows hasta la ventana del cliente Github y se incluirán en la lista de repositorios a la derecha.

Flujo de trabajo recomendado

Una vez instalado y configurado el cliente GitHub está listo para utilizarse en muchas tareas Git habituales. El flujo de trabajo previsto para esta herramienta se denomina "Flujo GitHub." Veremos con más detalle esto en El Flujo de Trabajo en GitHub, pero, en esencia, consistirá en (a) realizar commits sobre una rama y (b) sincronizarse con un repositorio remoto con cierta regularidad.

La gestión de ramas es uno de los puntos en donde las dos herramientas varían. En Mac, hay un botón en la parte superior de la ventana para crear una rama nueva:

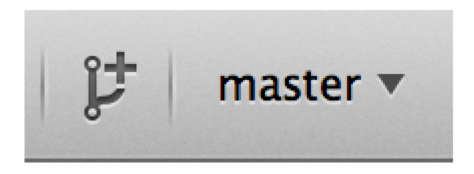


Figure 157. Botón “Create Branch” en Mac.

En Windows, esto se realiza escribiendo el nombre de la nueva rama en la componente de gestión de ramas:

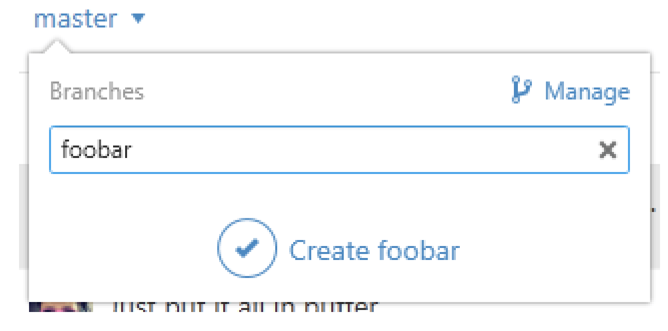


Figure 158. Creación de una rama en Windows.

Una vez creada la rama, realizar nuevos commits sobre ella es bastante sencillo. Realizas algunos cambios en el directorio de trabajo y, si vuelves a la ventana del cliente GitHub, te muestra los archivos que han sido modificados. Introduce un mensaje para el commit, seleccionas los archivos que quieres incluir y pulsas sobre el botón “Commit” (ctrl-enter o -enter).

La principal forma de interactuar con otros repositorios en la red es mediante la funcionalidad “Sync”. Internamente Git dispone de operaciones diferenciadas para mandar (push), recuperar (fetch), fusionar (merge) y reorganizar (rebase), aunque los clientes GitHub juntan todas ellas en una sola funcionalidad multi-paso. Esto es lo que ocurre cuando se pulsa el botón Sync:

1. `git pull --rebase`. Si esto falla es debido a conflictos en la fusión, echa mano de `git pull --no-rebase`.

1. `git push`.

Esta es la secuencia más frecuente de comandos de red cuando se trabaja de esta manera, así que meter estos comandos ahorra un montón de tiempo.

Resumen

Estas herramientas resultan ideales para el flujo de trabajo para el que fueron diseñadas. Tanto desarrolladores como no desarrolladores pueden ponerse a colaborar en un proyecto en cuestión de minutos y la mayoría de las buenas prácticas para esta forma de trabajo vienen integradas con las herramientas. Sin embargo, si el flujo de trabajo es distinto o si se quiere más control sobre cómo y dónde se hacen las operaciones con la red, quizás sea más conveniente algún otro cliente o la línea de comandos.

Otras herramientas gráficas

Existen otra serie de cliente Git con interfaz gráfica que oscilan entre las herramientas especializadas con un único fin y las aplicaciones que intentan presentar todo lo que Git puede

hacer. La página oficial de Git tiene una sucinta lista de los clientes más utilizados en http://git-scm.com/downloads/guis. Aunque en la wiki de Git se puede encontrar una lista más completa https://git.wiki.kernel.org/index.php/Interfaces,_frontends,_and_tools#Graphical_Interfaces.

Git en Visual Studio

Desde la versión Visual Studio 2013 Update 1, los usuarios de Visual Studio disponen de un cliente Git integrado en el IDE. Visual Studio había tenido funcionalidades integradas de control de versiones desde hacía tiempo pero estaban orientadas hacia sistemas centralizados con bloqueo de archivos, así que Git no se adecuaba bien a ese flujo de trabajo. La compatibilidad de Git en Visual Studio 2013 se ha apartado de esta antigua funcionalidad y el resultado es una adaptación mucho mejor entre Visual Studio y Git.

Para localizar esta funcionalidad, abre un proyecto que esté controlado mediante Git (o simplemente usa `git init` en un proyecto ya existente) y selecciona en el menú VIEW > Team Explorer. Puedes ver el visor de "Connect" (Conectar) que se parecerá un poco a ésta:

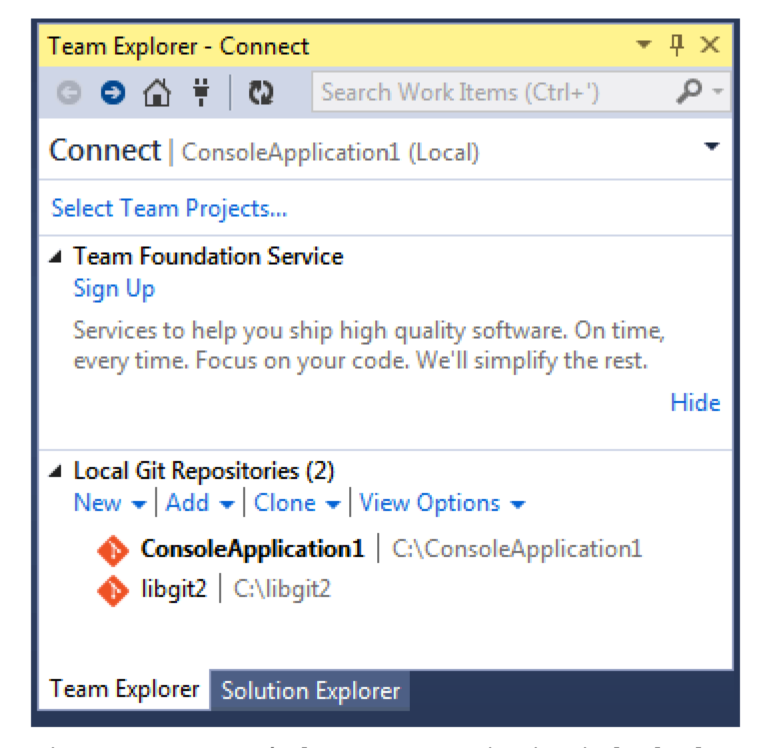


Figure 159. Conectándose a un repositorio Git desde el Team Explorer.

Visual Studio recuerda todos los proyectos que se han abierto y que están controlados mediante Git, y estarán disponibles en la lista de abajo. Si no consigues ver el proyecto, haz clic en el enlace "Add" y escribe la ruta del directorio de trabajo. Haciendo doble clic sobre uno de los repositorios locales Git, te lleva a la vista de inicio, que es como Vista de inicio del repositorio Git en Visual Studio.. Este es el centro para realizar las acciones Git. Cuando estás *escribiendo* código, probablemente dediques la mayor parte del tiempo sobre el visor de "Changes" (Cambios), aunque cuando llegue el momento de descargar (pull down) los cambios realizados por tus compañeros, seguramente utilizarás los visores de "Unsynced Commits" (Commit no sincronizados) y de "Branches" (Ramas).

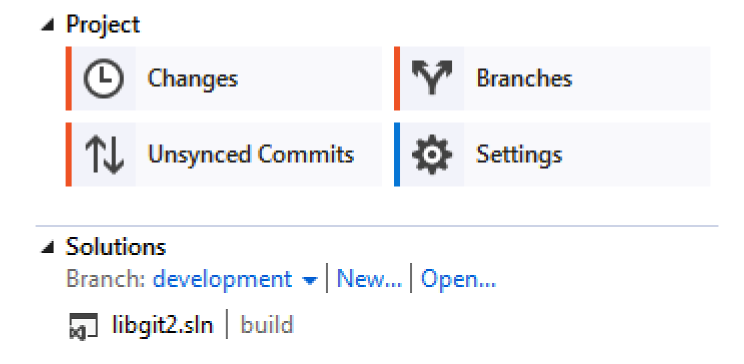


Figure 160. Vista de inicio del repositorio Git en Visual Studio.

Visual Studio tiene ahora un entorno gráfico para Git potente y orientado a tareas. Incluye un visor de históricos lineal, un visor de diferencias, comandos remotos y otras muchas funcionalidades. Puedes dirigirte a http://msdn.microsoft.com/en-us/library/hh850437.aspx para una documentación más completa de todas estas funcionalidades (que no cabrían en esta sección).

Git en Eclipse

Eclipse trae de serie una componente denominada Egit que proporciona una interfaz bastante completa de las operaciones con Git. Para acceder a ella, hay que ir a la perspectiva Git (en el menú Window > Open Perspective > Other…, y entonces seleccionar "Git").

Figure 161. Entorno EGit en Eclipse.

EGit tiene una completa documentación, a la que se puede acceder yendo a Help > Help Contents, y seleccionando el nodo "EGit Documentation" en el listado de contenidos.

Git con Bash

Si eres usuario de Bash, puedes acceder a una serie de características de la consola o terminal que pueden hacer mucho más llevadera la experiencia con Git. Aunque Git viene con extensiones para varios tipos de terminales, éstas no suelen estar activadas por defecto.

Lo primero es obtener una copia del archivo `contrib/completion/git-completion.bash` del código fuente de Git. Haz una copia de este archivo en cualquier lugar, por ejemplo el directorio de inicio, y añádelo al `.bashrc`:

```
. ~/git-completion.bash
```

Una vez hecho esto, cámbiate a un directorio que sea un repositorio git y teclea:

```
$ git chec<tab>
```

… Bash debería autocompletar con `git checkout`. Esto funciona con todos los subcomandos de Git, parámetros en línea de comandos y en nombres remotos y referencias, cuando sea apropiado.

También resulta útil personalizar el prompt para que muestre información sobre el repositorio Git que hay en el directorio actual. Se puede hacer tan simple o tan complejo como quieras aunque hay una serie de elementos de información que a la mayoría de las personas les resultan útiles, como la rama actual o el estado del directorio de trabajo. Para añadirlo al prompt, simplemente haz una copia del archivo `contrib/completion/git-prompt.sh` del código fuente de Git al directorio de inicio y añade lo siguiente al `.bashrc`:

```
. ~/git-prompt.sh
export GIT_PS1_SHOWDIRTYSTATE=1
export PS1='\w$(__git_ps1 " (%s)")\$ '
```

La `\w` indica que muestre el directorio de trabajo actual, la `\$` que muestre el símbolo `$` como parte del prompt y el `__git_ps1 " (%s)"` llama una función en `git-prompt.sh` con un parámetro de formato. Así, el prompt del bash tendrá este aspecto cuando estemos en un proyecto gestionado con Git:

Figure 162. Prompt personalizado en `bash`*.*

Ambos scripts tienen una práctica documentación, por lo que, para más información, revisa los contenidos de `git-completion.bash` y `git-prompt.sh`.

Git en Zsh

Git también viene con una librería de completación de pestañas para Zsh. Simplemente copie `contrib/completion/git-completion.zsh` a su directorio local y anclelo de su `.zshrc`. La interfaz de Zsh es un poco más poderosa que la de Bash:

```
$ git che<tab>
check-attr        -- display gitattributes information
check-ref-format  -- ensure that a reference name is well formed
checkout          -- checkout branch or paths to working tree
checkout-index    -- copy files from index to working directory
cherry            -- find commits not merged upstream
cherry-pick       -- apply changes introduced by some existing commits
```

Las completaciones de pestañas ambiguas no sólo son listadas; tienen descripciones muy útiles y puede navegar graficamente por la lista presionando tab repetidas veces. Esto funciona con comandos de Git, sus argumentos y nombres de cosas dentro del repositorio (como referencias y repositorios remotos), así como nombres de archivos y todas las otras cosas que Zhs sabe como "pestaña-completar".

Zsh resulta ser compatible con Bash cuando se trata de personalización de prompts, pero este le permite tener un prompt del lado derecho también. Para incluír el nombre del branch en el lado derecho, añada estas líneas a su archivo `~/.zshrc`:

```
setopt prompt_subst
. ~/git-prompt.sh
export RPROMPT=$'$(__git_ps1 "%s")'
```

Esto lleva a una muestra del branch actual en el lado a mano derecha de la ventana del terminal, siempre que tu caparazón esté dentro de un repositorio Git. Se ve un poco como esto:



Figure 163. Customized `zsh` prompt.

Zsh es lo suficientemente potente, tanto así que existen marcos metodológicos enteros dedicados a mejorarlo. Uno de estos se llama "oh-my-zsh", y puede ser encontrado en https://github.com/robbyrussell/oh-my-zsh. El sistema plug-in de oh-my-zsh viene con una poderosa completación de pestañas git, y tiene una variedad de "temas" de prompt, de los cuales muchos muestran datos de control de versiones. An example of an oh-my-zsh theme. es tan sólo un ejemplo de lo que puede realizarse con este sistema.

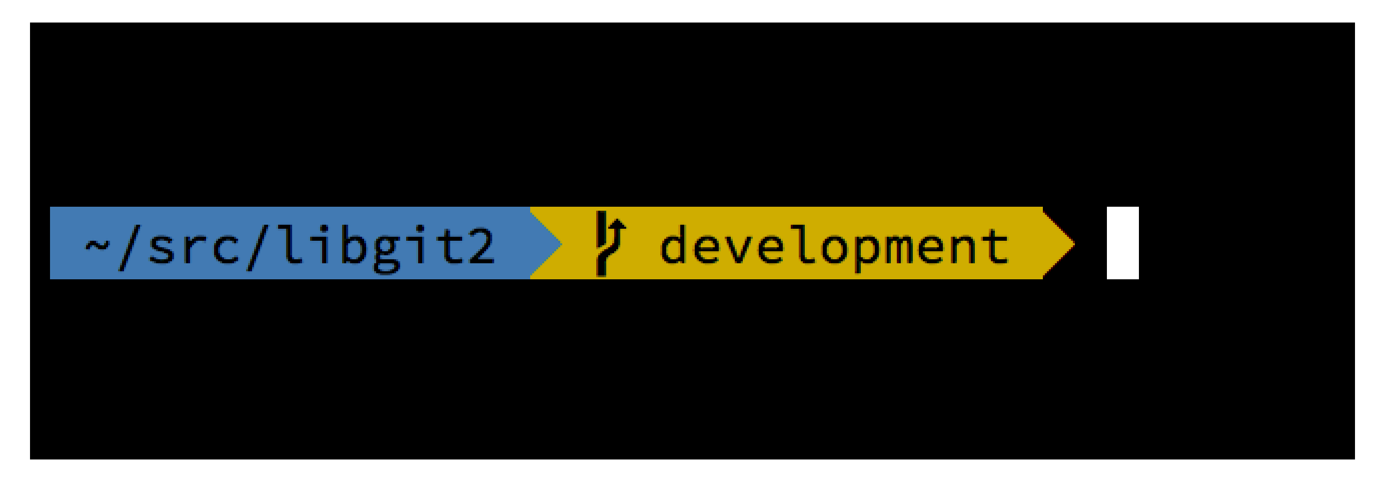


Figure 164. An example of an oh-my-zsh theme.

Git en Powershell

El terminal de la línea de comandos estándar en Windows (`cmd.exe`) no es realmente capaz de ofrecer una experiencia personalizada en Git, pero si está utilizando Powershell tiene mucha suerte. Un paquete llamado Posh-Git(https://github.com/dahlbyk/posh-git) proporciona comodidades poderosas para la completación de pestañas, así como un prompt mejorado para

ayudarle a mantenerse al tanto sobre el estado de su repositorio. Se ve de esta manera:



Figure 165. Powershell con Posh-git.

Si usted ha instalado Github para Windows, Posh-Git se encuentra incluído. Todo lo que tiene que hacer es añadir estas lineas a su `profile.ps1` (El cual se encuentra usualmente en `C:\Users\<username>\Documents\WindowsPowerShell`):

```
. (Resolve-Path "$env:LOCALAPPDATA\GitHub\shell.ps1")
. $env:github_posh_git\profile.example.ps1
```

Si no es un usuario de Github para Windows, simplemente descargue una versión de Posh-Git desde (https://github.com/dahlbyk/posh-git) y descomprimala en el directorio `WindowsPowershell`. Luego abra un prompt de Powershell como administrador y haga lo siguiente:

```
> Set-ExecutionPolicy RemoteSigned -Scope CurrentUser -Confirm
> cd ~\Documents\WindowsPowerShell\posh-git
> .\install.ps1
```

Esto añadirá la línea correspondiente a su archivo `profile.ps1` y posh-git estará activo la próxima vez que habra su prompt.

Resumen

Has aprendido a sacar partido a toda la potencia de Git desde dentro de la herramienta para poder usarlo en el tu trabajo diario así como a acceder a los repositorios Git desde tus propios programas.

Appendix A: Integrando Git en tus Aplicaciones

Si tu aplicación es para desarrolladores, es muy probable que pueda beneficiarse de la integración con el control de código fuente. Incluso las aplicaciones que no sean para desarrolladores, tales como editores de documentos, podrían beneficiarse de las características de control de versiones, y el modelo de Git funciona muy bien para muchos escenarios diferentes.

Si necesitas integrar Git con tu aplicación, tienes básicamente tres opciones: generar un shell y usar la herramienta de línea de comandos de Git; Libgit2; y JGit.

Git mediante Línea de Comandos

Una opción es generar un proceso shell y utilizar la herramienta de línea de comandos de Git para hacer el trabajo. Esto tiene la ventaja de ser canónico, y todas las características de Git están soportadas. Esto también resulta ser bastante fácil, ya que la mayoría de los entornos de ejecución tienen una forma relativamente sencilla para invocar un proceso con argumentos de la línea de comandos. Sin embargo, este enfoque tiene algunas desventajas.

Una es que toda la salida es un texto plano. Esto significa que tendrás que analizar el formato de salida cambiante de Git para leer la información de progreso y de resultado, lo que puede ser ineficiente y propenso a errores.

Otra es la falta de recuperación de errores. Si un repositorio está dañado de alguna manera, o el usuario tiene un valor de configuración con formato incorrecto, Git simplemente se negará a realizar muchas operaciones.

Otra más es la gestión de procesos. Git requiere que mantengas un entorno de shell en un proceso separado, lo que puede añadir complejidad no deseada. Tratar de coordinar muchos de estos procesos (especialmente cuando se accede potencialmente el mismo repositorio de varios procesos) puede ser todo un reto.

Libgit2

© Otra opción a tu disposición es utilizar Libgit2. Libgit2 es una implementación de Git libre de dependencias, con un enfoque en tener una buena API para su uso dentro de otros programas. Puedes encontrarla en http://libgit2.github.com.

En primer lugar, echemos un vistazo a la apariencia de la API C. He aquí una gira relámpago:

```
// Open a repository
git_repository *repo;
int error = git_repository_open(&repo, "/path/to/repository");

// Dereference HEAD to a commit
git_object *head_commit;
error = git_revparse_single(&head_commit, repo, "HEAD^{commit}");
git_commit *commit = (git_commit*)head_commit;

// Print some of the commit's properties
printf("%s", git_commit_message(commit));
const git_signature *author = git_commit_author(commit);
printf("%s <%s>\n", author->name, author->email);
const git_oid *tree_id = git_commit_tree_id(commit);

// Cleanup
git_commit_free(commit);
git_repository_free(repo);
```

El primer par de líneas abre un repositorio Git. El tipo `git_repository` representa un identificador a un repositorio con una caché en memoria. Éste es el método más simple, para cuando se conoce la ruta exacta al directorio de trabajo de un repositorio o carpeta `.git`. También está el `git_repository_open_ext` que incluye opciones para buscar, `git_clone` y compañía para hacer un clon local de un repositorio remoto, y `git_repository_init` para la creación de un repositorio completamente nuevo.

El segundo fragmento de código utiliza la sintaxis rev-parse (ver Referencias por rama para más información) para obtener el commit al HEAD finalmente apunta. El tipo devuelto es un puntero `git_object`, lo que representa algo que existe en la base de datos de objetos de Git para un repositorio. `git_object` es en realidad un tipo '*padre*' de varios tipos diferentes de objetos; el diseño de memoria para cada uno de los tipos '*hijo*' es el mismo que para `git_object`, por lo que puedes hacer casting de forma segura hacia la derecha. En este caso, `git_object_type (commit)` devolvería `GIT_OBJ_COMMIT`, así que es seguro hacer casting a un puntero `git_commit`.

El siguiente fragmento muestra cómo acceder a las propiedades del commit. La última línea aquí utiliza un tipo `git_oid`; esta es la representación de Libgit2 para un hash SHA-1.

De esta muestra, un par de patrones han comenzado a surgir:

- Si se declara un puntero y se pasa una referencia a él en una llamada Libgit2, la llamada devolverá probablemente un código de error entero. Un valor `0` indica éxito; cualquier otra cosa es un error.
- Si Libgit2 rellena un puntero para ti, eres responsable de liberarlo.
- Si Libgit2 devuelve un puntero `const` desde una llamada, no tienes que liberarlo, pero no será válido cuando el objeto al que pertenece sea liberado.
- Escribir C es un poco doloroso.

Esto último significa que no es muy probable que estés escribiendo C cuando utilices Libgit2.

Afortunadamente, hay una serie de vínculos específicos del lenguaje disponibles que hacen que sea bastante fácil trabajar con repositorios Git desde su entorno y lenguaje específico. Echemos un vistazo al ejemplo anterior escrito utilizando los vínculos de Ruby para Libgit2, que llevan el nombre Rugged, y se puede encontrar en https://github.com/libgit2/rugged.

```
repo = Rugged::Repository.new('path/to/repository')
commit = repo.head.target
puts commit.message
puts "#{commit.author[:name]} <#{commit.author[:email]}>"
tree = commit.tree
```

Como se puede ver, el código es mucho menos desordenado. En primer lugar, Rugged utiliza excepciones; puede elevar cosas como `ConfigError` o `ObjectError` para indicar condiciones de error. En segundo lugar, no hay liberación explícita de los recursos, ya que Ruby es recolector de basura. Echemos un vistazo a un ejemplo un poco más complicado: la elaboración de un commit desde cero

```
blob_id = repo.write("Blob contents", :blob) ①

index = repo.index
index.read_tree(repo.head.target.tree)
index.add(:path => 'newfile.txt', :oid => blob_id) ②

sig = {
    :email => "bob@example.com",
    :name => "Bob User",
    :time => Time.now,
}

commit_id = Rugged::Commit.create(repo,
    :tree => index.write_tree(repo), ③
    :author => sig,
    :committer => sig, ④
    :message => "Add newfile.txt", ⑤
    :parents => repo.empty? ? [] : [ repo.head.target ].compact, ⑥
    :update_ref => 'HEAD', ⑦
)
commit = repo.lookup(commit_id) ⑧
```

① Se crea un nuevo blob, que contiene el contenido de un nuevo archivo.

② Se rellena el index con el árbol de head commit, y añadimos el nuevo archivo a la ruta `newfile.txt`.

③ Esto crea un nuevo árbol en la ODB, y lo utiliza para un nuevo commit.

④ Utilizamos la misma firma, tanto para los campos del autor como del confirmador.

⑤ El mensaje del commit.

⑥ Al crear un commit, tienes que especificar los nuevos padres del commit. Éste utiliza la punta de HEAD para un padre único.

⑦ Rugged (y Libgit2) pueden actualizar opcionalmente una referencia al hacer un commit.

⑧ El valor de retorno es el hash SHA-1 de un nuevo objeto commit, que luego se puede utilizar para obtener un objeto `Commit`.

El código en Ruby es bonito y limpio, pero ya que Libgit2 está haciendo el trabajo pesado, este código se ejecutará bastante rápido, también. Si no eres un rubyista, tocamos algunos otros vínculos en Otros Vínculos (Bindings).

Funcionalidad Avanzada

Libgit2 tiene un par de capacidades que están fuera del ámbito del núcleo de Git. Un ejemplo es la conectividad: Libgit2 te permite proporcionar '*backends*' a medida para varios tipos de operaciones, por lo que puedes almacenar las cosas de una manera diferente a como hace el Git original. Libgit2 permite backends personalizados para la configuración, el almacenamiento de referencias, y la base de datos de objetos, entre otras cosas.

Echemos un vistazo a cómo funciona esto. El código siguiente se ha tomado del conjunto de ejemplos de backend proporcionados por el equipo de Libgit2 (que se puede encontrar en https://github.com/libgit2/libgit2-backends). Así es como se configura un backend personalizado para una base de datos de objetos:

```
git_odb *odb;
int error = git_odb_new(&odb); ①

git_odb_backend *my_backend;
error = git_odb_backend_mine(&my_backend, /*…*/); ②

error = git_odb_add_backend(odb, my_backend, 1); ③

git_repository *repo;
error = git_repository_open(&repo, "some-path");
error = git_repository_set_odb(odb); ④
```

(Ten en cuenta que los errores son capturados, pero no tratados. Esperamos que tu código sea mejor que el nuestro.)

① Se inicializa un '*frontend*' a una base de datos de objetos (ODB), que actuará como contenedor de los '*backends*', que son los que hacen el trabajo real.

② Se inicializa un backend ODB personalizado.

③ Se añade el backend al frontend.

④ Se abre un repositorio, y se configura para que use nuestra ODB para buscar objetos.

Pero, ¿qué es esta cosa `git_odb_backend_mine`? Bien, ese es el constructor para tu propia implementación ODB, y puedes hacer lo que quieras allí, siempre y cuando rellenes en el `git_odb_backend` la estructura correctamente. A esto es a lo que *podría* parecerse:

```
typedef struct {
    git_odb_backend parent;

    // Some other stuff
    void *custom_context;
} my_backend_struct;

int git_odb_backend_mine(git_odb_backend **backend_out, /*…*/)
{
    my_backend_struct *backend;

    backend = calloc(1, sizeof (my_backend_struct));

    backend->custom_context = …;

    backend->parent.read = &my_backend__read;
    backend->parent.read_prefix = &my_backend__read_prefix;
    backend->parent.read_header = &my_backend__read_header;
    // …

    *backend_out = (git_odb_backend *) backend;

    return GIT_SUCCESS;
}
```

La restricción más sutil aquí es que el primer miembro de `my_backend_struct` debe ser una estructura `git_odb_backend`; esto asegura que la disposición de memoria sea la que el código Libgit2 espera. El resto es arbitrario; esta estructura puede ser tan grande o tan pequeña como necesites que sea.

La función de inicialización reserva memoria para la estructura, establece el contexto personalizado, y luego rellena los miembros de la estructura `parent` que soporta. Echa un vistazo al archivo `include/git2/sys/odb_backend.h` en el código fuente de Libgit2 para un conjunto completo de llamadas; tu caso de uso particular te ayudará a determinar cuál de éstas querrás soportar.

Otros Vínculos (Bindings)

Libgit2 tiene vínculos para muchos lenguajes. A continuación mostramos un pequeño ejemplo que usa algunos de los paquetes de vínculos más completos a fecha de este escrito; existen bibliotecas para muchos otros idiomas, incluyendo C++, Go, Node.js, Erlang, y la JVM, todos en diferentes etapas de madurez. La colección oficial de vínculos se puede encontrar navegando por los repositorios en https://github.com/libgit2. El código que escribiremos devolverá el mensaje del commit finalmente apuntado por HEAD (algo así como `git log -1`).

LibGit2Sharp

Si estás escribiendo una aplicación .NET o Mono, LibGit2Sharp (https://github.com/libgit2/libgit2sharp) es lo que estás buscando. Los vínculos están escritos en C#, y se ha tenido gran cuidado de envolver las llamadas a Libgit2 crudo con APIs CLR de apariencia nativa. Esta es la

apariencia de nuestro programa de ejemplo:

```
new Repository(@"C:\path\to\repo").Head.Tip.Message;
```

Para las aplicaciones de escritorio de Windows, incluso hay un paquete NuGet que le ayudará a empezar rápidamente.

objective-git

Si la aplicación se ejecuta en una plataforma de Apple, es muy probable que use Objective-C como su lenguaje de implementación. Objective-Git (https://github.com/libgit2/objective-git) es el nombre de los vínculos Libgit2 para ese entorno. El programa de ejemplo es el siguiente:

```
GTRepository *repo =
    [[GTRepository alloc] initWithURL:[NSURL fileURLWithPath: @"/path/to/repo"]
error:NULL];
NSString *msg = [[[repo headReferenceWithError:NULL] resolvedTarget] message];
```

Objective-git es totalmente interoperable con Swift, así que no temas, si has dejado atrás Objective-C.

pygit2

Los vínculos para Libgit2 en Python se llaman Pygit2, y se pueden encontrar en http://www.pygit2.org/. Nuestro programa de ejemplo:

```
pygit2.Repository("/path/to/repo") # open repository
    .head                          # get the current branch
    .peel(pygit2.Commit)           # walk down to the commit
    .message                       # read the message
```

Otras Lecturas

Por supuesto, un tratamiento completo de las capacidades de Libgit2 está fuera del alcance de este libro. Si deseas más información sobre Libgit2 en sí mismo, hay documentación de la API en https://libgit2.github.com/libgit2, y un conjunto de guías en https://libgit2.github.com/docs.

Para otros vínculos (bindings), comprobar el README incorporado y los tests; a menudo hay pequeños tutoriales y enlaces a otras lecturas allí.

JGit

Si deseas utilizar Git desde dentro de un programa Java, hay una biblioteca Git completamente funcional llamada JGit. JGit es una implementación relativamente completa de Git escrita de forma nativa en Java, y que se utiliza ampliamente en la comunidad Java. El proyecto JGit está bajo el paraguas de Eclipse, y su "casa" puede encontrarse en http://www.eclipse.org/jgit.

Getting Set Up

Hay varias formas de conectar tu proyecto con JGit y empezar a escribir código usando éste. Probablemente la más fácil sea utilizar Maven -la integración se consigue añadiendo el siguiente fragmento a la etiqueta `<dependencies>` en tu archivo pom.xml:

```
<dependency>
    <groupId>org.eclipse.jgit</groupId>
    <artifactId>org.eclipse.jgit</artifactId>
    <version>3.5.0.201409260305-r</version>
</dependency>
```

La `version` es bastante probable que habrá avanzado para el momento en que leas esto; comprueba http://mvnrepository.com/artifact/org.eclipse.jgit/org.eclipse.jgit para obtener información actualizada del repositorio. Una vez que se realiza este paso, Maven automáticamente adquirirá y utilizará las bibliotecas JGit que necesites.

Si prefieres gestionar las dependencias binarias tú mismo, binarios JGit pre-construidos están disponibles en http://www.eclipse.org/jgit/download. Puedes construirlos en tu proyecto ejecutando un comando como el siguiente:

```
javac -cp .:org.eclipse.jgit-3.5.0.201409260305-r.jar App.java
java -cp .:org.eclipse.jgit-3.5.0.201409260305-r.jar App
```

Fontanería

JGit tiene dos niveles básicos de la API: fontanería y porcelana. La terminología de éstos proviene de Git, y JGit se divide en más o menos los mismos tipos de áreas: las API de porcelana son un front-end amigable para las acciones comunes a nivel de usuario (el tipo de cosas para las que un usuario normal utilizaría la herramienta de línea de comandos de Git), mientras que las API de fontanería son para interactuar directamente a bajo nivel con los objetos del repositorio.

El punto de partida para la mayoría de las sesiones JGit es la clase `Repository`, y la primera cosa que querrás hacer es crear una instancia de la misma. Para un repositorio basado en sistema de archivos (sí, JGit permite otros modelos de almacenamiento), esto se logra utilizando `FileRepositoryBuilder`:

```
// Create a new repository; the path must exist
Repository newlyCreatedRepo = FileRepositoryBuilder.create(
    new File("/tmp/new_repo/.git"));

// Open an existing repository
Repository existingRepo = new FileRepositoryBuilder()
    .setGitDir(new File("my_repo/.git"))
    .build();
```

El constructor tiene una API fluida para proporcionar todo lo que necesitas para encontrar un repositorio Git, tanto si tu programa sabe exactamente donde se encuentra como si no. Puede utilizar variables de entorno ((`.readEnvironment()`), empezar a partir de un lugar en el directorio de trabajo y buscar (`.setWorkTree(…).findGitDir()`), o simplemente abrir un directorio `.git` conocido como más arriba.

Una vez que tengas una instancia `Repository`, se pueden hacer todo tipo de cosas con ella. He aquí una muestra rápida:

```
// Get a reference
Ref master = repo.getRef("master");

// Get the object the reference points to
ObjectId masterTip = master.getObjectId();

// Rev-parse
ObjectId obj = repo.resolve("HEAD^{tree}");

// Load raw object contents
ObjectLoader loader = repo.open(masterTip);
loader.copyTo(System.out);

// Create a branch
RefUpdate createBranch1 = repo.updateRef("refs/heads/branch1");
createBranch1.setNewObjectId(masterTip);
createBranch1.update();

// Delete a branch
RefUpdate deleteBranch1 = repo.updateRef("refs/heads/branch1");
deleteBranch1.setForceUpdate(true);
deleteBranch1.delete();

// Config
Config cfg = repo.getConfig();
String name = cfg.getString("user", null, "name");
```

Hay bastantes cosas que suceden aquí, así que vamos a examinarlo sección a sección.

La primera línea consigue un puntero a la referencia `master`. JGit obtiene automáticamente la referencia master *real*, que reside en `refs/heads/master`, y devuelve un objeto que te permite obtener información acerca de la referencia. Puedes obtener el nombre (`.getName()`), y también el objeto destino de una referencia directa (`.getObjectId()`) o la referencia a la que apunta mediante una referencia simbólica (`.getTarget()`). Los objetos Ref también se utilizan para representar referencias a etiquetas y objetos, por lo que puedes preguntar si la etiqueta está *'pelada'*, lo que significa que apunta al objetivo final de una (potencialmente larga) cadena de texto de objetos etiqueta.

La segunda línea obtiene el destino de la referencia `master`, que se devuelve como una instancia ObjectId. ObjectId representa el hash SHA-1 de un objeto, que podría o no existir en la base de datos

de objetos de Git. La tercera línea es similar, pero muestra cómo maneja JGit la sintaxis rev-parse (para más información sobre esto, consulta Referencias por rama); puedes pasar cualquier especificador de objeto que Git entienda, y JGit devolverá una ObjectId válida para ese objeto, o `null`.

Las dos líneas siguientes muestran cómo cargar el contenido en bruto de un objeto. En este ejemplo, llamamos a `ObjectLoader.copyTo()` para transmitir el contenido del objeto directamente a la salida estándar, pero ObjectLoader también tiene métodos para leer el tipo y el tamaño de un objeto, así como devolverlo como un array de bytes. Para objetos grandes (donde `.isLarge()` devuelve true), puedes llamar a `.openStream()` para obtener un objeto similar a InputStream del cual puedes leer los datos del objeto en bruto si almacenarlo en memoria en seguida.

Las siguientes líneas muestran lo que se necesita para crear una nueva rama. Creamos una instancia RefUpdate, configuramos algunos parámetros, y llamamos a `.update()` para activar el cambio. Inmediatamente después de esto está el código para eliminar esa misma rama. Ten en cuenta que se requiere `.setForceUpdate(true)` para que esto funcione; de lo contrario la llamada `.delete()` devolverá `REJECTED`, y no pasará nada.

El último ejemplo muestra cómo buscar el valor `user.name` a partir de los archivos de configuración de Git. Este ejemplo de configuración utiliza el repositorio que abrimos anteriormente para la configuración local, pero detectará automáticamente los archivos de configuración global y del sistema y leerá los valores de ellos también.

Ésta es sólo una pequeña muestra de la API de fontanería completa; hay muchos más métodos y clases disponibles. Tampoco se muestra aquí la forma en la que JGit maneja los errores, que es a través del uso de excepciones. La API de JGit a veces lanza excepciones Java estándar (como `IOException`), pero también hay una gran cantidad de tipos de excepciones específicas de JGit que se proporcionan (tales como `NoRemoteRepositoryException`, `CorruptObjectException`, y `NoMergeBaseException`).

Porcelana

Las APIs de fontanería son bastante completas, pero puede ser engorroso encadenarlas juntas para alcanzar objetivos comunes, como la adición de un archivo en el index, o hacer un nuevo commit. JGit proporciona un conjunto de APIs de más alto nivel para facilitar esto, y el punto de entrada a estas APIs es la clase `Git`:

```
Repository repo;
// construct repo...
Git git = new Git(repo);
```

La clase Git tiene un buen conjunto de métodos estilo *builder* de alto nivel que se pueden utilizar para construir un comportamiento bastante complejo. Echemos un vistazo a un ejemplo - haciendo algo como `git ls-remote`:

```
CredentialsProvider cp = new UsernamePasswordCredentialsProvider("username",
"p4ssw0rd");
Collection<Ref> remoteRefs = git.lsRemote()
    .setCredentialsProvider(cp)
    .setRemote("origin")
    .setTags(true)
    .setHeads(false)
    .call();
for (Ref ref : remoteRefs) {
    System.out.println(ref.getName() + " -> " + ref.getObjectId().name());
}
```

Este es un patrón común con la clase Git; los métodos devuelven un objeto de comando que te permite encadenar llamadas a métodos para establecer los parámetros, que se ejecutan cuando se llama `.call()`. En este caso, estamos solicitando las etiquetas del repositorio remoto `origin`, pero no las cabezas (heads). Observa también el uso de un objeto `CredentialsProvider` para la autenticación.

Muchos otros comandos están disponibles a través de la clase Git, incluyendo, aunque no limitado, a `add`, `blame`, `commit`, `clean`, `push`, `rebase`, `revert`, y `reset`.

Otras Lecturas

Esta es sólo una pequeña muestra de todas las posibilidades de JGit. Si estás interesado y deseas aprender más, aquí tienes dónde buscar información e inspiración:

- La documentación API oficial de JGit está disponible en línea en http://download.eclipse.org/jgit/docs/latest/apidocs. Estos son Javadoc estándar, por lo que tu IDE JVM favorito será capaz de instalarlos de forma local, también.
- El "libro de cocina" de JGit en https://github.com/centic9/jgit-cookbook tiene muchos ejemplos de cómo realizar tareas específicas con JGit.
- Hay varios buenos recursos indicados en http://stackoverflow.com/questions/6861881.

Appendix B: Comandos de Git

A lo largo del libro hemos introducido docenas de comandos de Git y nos hemos esforzado para introducirlos dentro de una especie de narrativa, añadiendo más comandos a la historia poco a poco. Sin embargo, esto nos deja con ejemplos de uso de los comandos algo dispersos por todo el libro.

En este apéndice, repasaremos todos los comandos de Git que hemos tratado a lo largo del libro, agrupados por el uso que se les ha dado. Vamos a hablar de lo que hace cada comando de manera muy general y a continuación señalaremos en qué parte del libro puedes encontrar un uso de él.

Configuración

Hay dos comandos que se usan bastante, desde las primeras invocaciones de Git hasta el ajuste y referenciamiento diario, los comandos `config` y `help`.

git config

Git tiene una forma predeterminada de hacer cientos de cosas. Para muchas de estas cosas, puedes indicar a Git hacerlas por defecto de una manera diferente, o establecer tus preferencias. Esto incluye todo, desde decir a Git cuál es tu nombre a las preferencias de color específicas del terminal o qué editor utilizar. Hay varios archivos desde los que este comando lee y a los que escribe, así puedes establecer los valores a nivel global o hacia abajo para repositorios específicos.

El comando `git config` se ha utilizado en casi todos los capítulos del libro.

En Configurando Git por primera vez lo usamos para especificar nuestro nombre, dirección de correo electrónico y editor de preferencia antes incluso de comenzar a utilizar Git.

En Alias de Git mostramos cómo se puede utilizar para crear comandos abreviados que se expanden para secuencias largas de opciones para no tener que escribirlas cada vez.

En Reorganizar el Trabajo Realizado lo usamos para hacer `--rebase` predeterminado cuando se ejecuta `git pull`.

En Almacenamiento de credenciales lo usamos para establecer un almacén predeterminado para tus contraseñas HTTP.

En Expansión de palabras clave mostramos como configurar filtros sucios y limpios sobre contenido que entra y sale de Git.

Finalmente, básicamente la totalidad de Configuración de Git está dedicado al comando.

git help

El comando `git help` se utiliza para presentarte toda la documentación contenida con Git sobre cualquier comando. A pesar de que estamos dando una visión general de la mayoría de los más populares en este apéndice, para obtener una lista completa de todas las posibles opciones e indicadores para cada orden, siempre se puede ejecutar `git help <command>`.

Introducimos el comando `git help` en ¿Cómo obtener ayuda? y te mostramos cómo utilizarlo para encontrar más información sobre el `git shell` en Configurando el servidor.

Obtener y Crear Proyectos

Hay dos maneras de obtener un repositorio Git. Una de ellas es copiarlo desde un repositorio existente en la red o en otro lugar y la otra es crear uno nuevo en un directorio existente.

git init

Para tomar un directorio y convertirlo en un nuevo repositorio Git en el que puedas empezar a controlar sus versiones, simplemente puedes ejecutar `git init`.

En primer lugar, introducimos esto en Obteniendo un repositorio Git, donde mostramos la creación de un nuevo repositorio para empezar a trabajar.

Hablamos brevemente acerca de cómo puedes cambiar la rama por defecto desde "master" en Ramas Remotas.

Usamos este comando para crear un repositorio desnudo (bare) vacío para un servidor en Colocando un Repositorio Vacío en un Servidor.

Por último, examinamos algunos de los detalles de lo que realmente hace detrás de escena en Fontanería y porcelana.

git clone

El comando 'git clone` es en realidad una especie de envoltura alrededor de varios otros comandos. Éste crea un nuevo directorio, entra en él y ejecuta `git init` para que sea un repositorio vacío de Git, añade uno remoto (`git remote add`) hacia la dirección URL que se le pasa (por defecto llamado `origin`), ejecuta un `git fetch` de ese repositorio remoto y después activa el último commit en el directorio de trabajo con `git checkout`.

El comando `git clone` es utilizado en docenas de lugares a lo largo del libro, pero sólo enumeraremos algunos lugares interesantes.

Básicamente se introdujo y se explicó en Clonando un repositorio existente, donde examinamos algunos ejemplos.

En Configurando Git en un servidor nos fijamos en el uso de la opción `--bare` para crear una copia de un repositorio Git sin directorio de trabajo.

En Agrupaciones lo usamos para desempaquetar un repositorio Git empaquetado (bundle).

Finalmente, en Clonación de un Proyecto con Submódulos aprendemos la opción `--recursive` para realizar la clonación de un repositorio con submódulos un poco más simple.

Aunque se usa en muchos otros lugares a través del libro, estos son los que son algo únicos o donde se utiliza en formas que son un poco diferentes.

Seguimiento Básico

Para el flujo de trabajo básico de la preparación de contenido y su confirmación a su historia, hay sólo unos pocos comandos básicos.

git add

El comando `git add` añade contenido del directorio de trabajo al área de ensayo (staging area o '*index*') para la próxima confirmación. Cuando se ejecuta el comando `git commit`, éste, de forma predeterminada, sólo mira en esta área de ensayo, por lo que `git add` se utiliza para fabricar exactamente lo que te gustaría fuese tu próxima instantánea a confirmar.

Este comando es un comando increíblemente importante en Git y se menciona o se utiliza docenas de veces en este libro. Vamos a cubrir rápidamente algunos de los usos únicos que se pueden encontrar.

En primer lugar, introducimos y explicamos `git add` en detalle en Rastrear Archivos Nuevos.

Mostramos como usarlo para resolver conflictos de fusión en Principales Conflictos que Pueden Surgir en las Fusiones.

Repasamos su uso para seguir de forma interactiva sólo partes específicas de un archivo modificado en Organización interactiva.

Por último, lo emulamos en un bajo nivel en Objetos tipo árbol, por lo que podemos tener una idea de lo que está haciendo detrás de escena.

git status

El comando `git status` te mostrará los diferentes estados de los archivos en tu directorio de trabajo y área de ensayo. Qué archivos están modificados y sin seguimiento y cuáles con seguimiento pero no confirmados aún. En su forma normal, también te mostrará algunos consejos básicos sobre cómo mover archivos entre estas etapas.

Primero cubrimos `status` en Revisando el Estado de tus Archivos, tanto en su forma básica como simplificada. Mientras lo utilizamos a lo largo del libro, prácticamente todo lo que puedes hacer con el comando `git status` está cubierto allí.

git diff

El comando `git diff` se utiliza cuando deseas ver las diferencias entre dos árboles. Esto prodría ser la diferencia entre tu entorno de trabajo y tu área de ensayo (`git diff` por sí mismo), entre tu área de ensayo y tu última confirmación o commit (`git diff --staged`), o entre dos confirmaciones (`git diff master branchB`).

En primer lugar, vemos el uso básico de `git diff` en Ver los Cambios Preparados y No Preparados, donde mostramos cómo ver que cambios tienen seguimiento y cuáles no tienen seguimiento aún.

Lo usamos para buscar posibles problemas con espacios en blanco antes de confirmar con la opción `--check` en Pautas de confirmación.

Vemos cómo comprobar las diferencias entre ramas de manera más eficaz con la sintaxis `git diff A...B` en Decidiendo qué introducir.

Lo usamos para filtrar diferencias en espacios en blanco con `-w` y como comparar diferentes etapas de archivos en conflicto con `--theirs`, `--ours` y `--base` en Fusión Avanzada.

Finalmente, lo usamos para realmente comparar cambios en submódulos con `--submodule` en Comenzando con los Submódulos.

git difftool

El comando `git difftool` simplemente lanza una herramienta externa para mostrar la diferencia entre dos árboles, en caso de que desees utilizar algo que no sea el comando`git diff` incorporado.

Mencionamos sólo brevemente esto en Ver los Cambios Preparados y No Preparados.

git commit

El comando `git commit` toma todos los contenidos de los archivos a los que se les realiza el seguimiento con `git add` y registra una nueva instantánea permanente en la base de datos y luego avanza el puntero de la rama en la rama actual.

En primer lugar, cubrimos los fundamentos en Confirmar tus Cambios. Allí también mostramos cómo utilizar el indicador `-a` para saltarse el paso `git add` en los flujos de trabajo diarios y cómo utilizar el indicador `-m` para pasar un mensaje de confirmación en la línea de comandos en lugar de lanzar un editor.

En Deshacer Cosas cubrimos el uso de la opción `--amend` para deshacer el commit más reciente.

En ¿Qué es una rama?, entramos en mucho mayor detalle en lo que `git commit` hace y por qué lo hace así.

Consideramos como firmar criptográficamente commits con el indicador `-S` en Firmando Commits.

Finalmente, echamos un vistazo a lo que el comando `git commit` hace en segundo plano y cómo se implementa realmente en Objetos de confirmación de cambios.

git reset

El comando `git reset` se utiliza sobre todo para deshacer las cosas, como posiblemente puedes deducir por el verbo. Se mueve alrededor del puntero `HEAD` y opcionalmente cambia el `index` o área de ensayo y también puede cambiar opcionalmente el directorio de trabajo si se utiliza `--hard`. Esta última opción hace posible que este comando pueda perder tu trabajo si se usa incorrectamente, por lo que asegúrese de entenderlo antes de usarlo.

En primer lugar, cubrimos efectivamente el uso más simple de `git reset` en Deshacer un Archivo Preparado, donde lo usamos para dejar de hacer seguimiento (unstage) de un archivo sobre el que habíamos ejecutado `git add`.

A continuación, lo cubrimos con bastante detalle en Reiniciar Desmitificado, que está

completamente dedicado a la explicación de este comando.

Utilizamos `git reset --hard` para abortar una fusión en Abortar una Fusión, donde también usamos `git merge --abort`, el cual es una especie de envoltorio para el comando `git reset`.

git rm

El comando `git rm` se utiliza para eliminar archivos del área de ensayo y el directorio de trabajo para Git. Es similar a `git add` en que pone en escena una eliminación de un archivo para la próxima confirmación.

Cubrimos el comando `git rm` con cierto detalle en Eliminar Archivos, incluyendo la eliminación de archivos de forma recursiva y sólo la eliminación de archivos desde el área de ensayo, pero dejándolos en el directorio de trabajo con `--cached`.

El único otro uso diferente de `git rm` en el libro está en Borrando objetos, donde utilizamos brevemente y explicamos el `--ignore-unmatch` al ejecutar `git filter-branch`, el cual simplemente hace que no salga un error cuando el archivo que estamos tratando de eliminar no existe. Esto puede ser útil para fines de scripting.

git mv

El comando `git mv` es un comando de conveniencia para mover un archivo y luego ejecutar `git add` en el nuevo archivo y `git rm` en el archivo antiguo.

Sólo mencionamos brevemente este comando en Cambiar el Nombre de los Archivos.

git clean

El comando `git clean` se utiliza para eliminar archivos no deseados de tu directorio de trabajo. Esto podría incluir la eliminación de artefactos de construcción temporal o la fusión de archivos en conflicto.

Cubrimos muchas de las opciones y escenarios en los que es posible usar el comando clean en Limpiando tu Directorio de Trabajo.

Ramificar y Fusionar

Hay sólo un puñado de comandos que implementan la mayor parte de la funcionalidad de ramificación y fusión en Git.

git branch

El comando `git branch` es en realidad una especie de herramienta de gestión de ramas. Puede listar las ramas que tienes, crear una nueva rama, eliminar ramas y cambiar el nombre de las ramas.

La mayor parte de Ramificaciones en Git está dedicada al comando `branch` y es utilizado a lo largo de todo el capítulo. En primer lugar, lo introducimos en Crear una Rama Nueva y examinamos la mayor parte de sus otras características (listar y borrar) en Gestión de Ramas.

En Hacer Seguimiento a las Ramas usamos la opción `git branch -u` para establecer una rama de seguimiento.

Finalmente, examinamos algo de lo que hace en segundo plano en Referencias Git.

git checkout

El comando `git checkout` se usa para cambiar de rama y revisar el contenido de tu directorio de trabajo.

En primer lugar, encontramos el comando en Cambiar de Rama junto con el comando `git branch`.

Vemos cómo usarlo para iniciar el seguimiento de ramas con el indicador `--track` en Hacer Seguimiento a las Ramas.

Lo usamos para reintroducir los conflictos de archivos con `--conflict=diff3` en Revisando Los Conflictos.

Entramos en más detalle sobre su relación con `git reset` en Reiniciar Desmitificado.

Finalmente, examinamos algún detalle de implementación en La CABEZA (HEAD).

git merge

La herramienta `git merge` se utiliza para fusionar uno o más ramas dentro de la rama que tienes activa. A continuación avanzará la rama actual al resultado de la fusión.

El comando `git merge` fue introducido por primera en Procedimientos Básicos de Ramificación. A pesar de que se utiliza en diversos lugares en el libro, hay muy pocas variaciones del comando `merge` — en general, sólo `git merge <branch>` con el nombre de la rama individual que se desea combinar.

Cubrimos cómo hacer una fusión aplastada (squashed merge) (donde Git fusiona el trabajo, pero finge como si fuera simplemente un nuevo commit sin registrar la historia de la rama que se está fusionando) al final de Proyecto público bifurcado.

Repasamos mucho sobre el proceso de fusión y dirección, incluyendo el comando `-Xignore-all-whitespace` y el indicador `--abort` para abortar un problema de fusión en Fusión Avanzada.

Aprendimos cómo verificar las firmas antes de la fusión si tu proyecto está usando fimas GPG en Firmando Commits.

Finalmente, aprendimos sobre la fusión de subárboles en Convergencia de Subárbol.

git mergetool

El comando `git mergetool` simplemente lanza un ayudante de fusión externo en caso de tener problemas con una combinación en Git.

Lo mencionamos rápidamente en Principales Conflictos que Pueden Surgir en las Fusiones y entramos en detalle sobre cómo implementar tu propia herramienta de fusión externa en

Herramientas externas para fusión y diferencias.

git log

El comando `git log` se utiliza para mostrar la historia registrada alcanzable de un proyecto desde la más reciente instantánea confirmada hacia atrás. Por defecto sólo se mostrará la historia de la rama en la que te encuentres, pero pueden ser dadas diferentes e incluso múltiples cabezas o ramas desde la que hacer el recorrido. También se utiliza a menudo para mostrar las diferencias entre dos o más ramas a nivel de commit.

Este comando se utiliza en casi todos los capítulos del libro para mostrar la historia de un proyecto.

Introducimos el comando y lo cubrimos con cierta profundidad en Ver el Historial de Confirmaciones. Allí vemos las opciones `-p` y `--stat` para tener una idea de lo que fue introducido en cada commit y las opciones `--pretty` y `--oneline` para ver el historial de forma más concisa, junto con unas simples opciones de filtrado de fecha y autor.

En Crear una Rama Nueva lo utilizamos con la opción `--decorate` para visualizar fácilmente donde se encuentran nuestros punteros de rama y también utilizamos la opción `--graph` para ver la apariencia de las historias divergentes.

En Pequeño equipo privado y Rangos de Commits cubrimos la sintaxis `branchA..branchB` al usar el comando `git log` para ver que commits son únicos a una rama en relación a otra rama. En Rangos de Commits repasamos esto bastante extensamente.

En Registro de Fusión y Tres puntos cubrimos el uso del formato `branchA...branchB` y la sintaxis `--left-right` para ver que está en una rama o en la otra pero no en ambas. En Registro de Fusión también vemos como utilizar la opción `--merge` para ayudarnos con la depuración de conflictos de fusión así como el uso de la opción `--cc` para ver conflictos de fusión en tu historia.

En Nombres cortos de RefLog usamos la opción `-g` para ver el reflog de Git a través de esta herramienta en lugar de hacer un recorrido de la rama.

En Buscando vemos el uso de las opciones `-S` y `-L` para hacer búsquedas bastante sofisticados de algo que sucedió históricamente en el código como ver la historia de una función.

En Firmando Commits vemos como usar `--show-signature` para añadir una cadena de texto de validación a cada commit en la salida de `git log` basado en si fue válidadmente firmado o no.

git stash

El comando `git stash` se utiliza para almacenar temporalmente el trabajo no confirmado con el fin de limpiar el directorio de trabajo sin tener que confirmar el trabajo no acabado en una rama.

Básicamente esto es enteramente cubierto en Guardado rápido y Limpieza.

git tag

El comando `git tag` se utiliza para dar un marcador permanente a un punto específico en el historial del código fuente. Generalmente esto se utiliza para cosas como las liberaciones (releases).

Este comando se introduce y se trata en detalle en Etiquetado y lo usamos en la práctica en Etiquetando tus versiones.

También cubrimos cómo crear una etiqueta con firma GPG tag con el indicador `-s` y verificamos uno con el indicador `-v` en Firmando tu trabajo.

Compartir y Actualizar Proyectos

No son muy numerosos los comandos de Git que acceden a la red, casi todos los comandos operan sobre la base de datos local. Cuando estas listo para compartir tu trabajo u obtener cambios de otros lugares, hay un puñado de comandos que tienen que ver con los repositorios remotos.

git fetch

El comando `git fetch` comunica con un repositorio remoto y obtiene toda la información que se encuentra en ese repositorio que no está en el tuyo actual y la almacena en tu base de datos local.

En primer lugar, observamos este comando en Traer y Combinar Remotos y seguimos viendo ejemplos de su uso en Ramas Remotas.

También lo usamos en varios de los ejemplos en Contribuyendo a un Proyecto.

Lo usamos para buscar una única referencia específica que se encuentra fuera del espacio por defecto en Referencias de Pull Request y vemos cómo buscar en una agrupación en Agrupaciones.

Configuramos refspecs altamente personalizadas con el fin de hacer que `git fetch` haga algo un poco diferente de lo predeterminado en Las especificaciones para hacer referencia a… (refspec).

git pull

El comando `git pull` es básicamente una combinación de los comandos `git fetch` y `git merge`, donde Git descargará desde el repositorio remoto especificado y a continuación, de forma inmediata intentará combinarlo en la rama en la que te encuentres.

Se introduce rápidamente en Traer y Combinar Remotos y mostramos la forma de ver lo que se fusionará si se ejecuta en Inspeccionar un Remoto.

También vemos cómo usarlo para ayudar con dificultades de rebase en Reorganizar una Reorganización.

Mostramos cómo usarlo con una URL para obtener los cambios de forma de una sola vez en Recuperando ramas remotas.

Por último, mencionamos muy rápidamente que se puede utilizar la opción `--verify-signatures` con el fin de verificar qué commits que estás descargando han sido firmados con GPG en <Firmando Commits.

git push

El comando `git push` se utiliza para comunicar con otro repositorio, calcular lo que tu base de datos

local tiene que la remota no tiene, y luego subir (push) la diferencia al otro repositorio. Se requiere acceso de escritura al otro repositorio y por tanto normalmente se autentica de alguna manera.

En primer lugar, observamos el comando `git push` en Enviar a Tus Remotos. Aquí cubrimos los fundamentos de subir una rama a un repositorio remoto. En Publicar profundizamos un poco más en la subida de ramas específicas y en Hacer Seguimiento a las Ramas vemos cómo configurar el seguimiento de ramas a las que subir automáticamente. En Eliminar Ramas Remotas utilizamos el indicador `--delete` para eliminar una rama en el servidor con `git push`.

A lo largo de Contribuyendo a un Proyecto vemos varios ejemplos de uso de `git push` para compartir trabajo en ramas de múltiples repositorios remotos.

Vemos cómo usarlo para compartir las etiquetas que has creado con la opción `--tags` en Compartir Etiquetas.

En Publicando Cambios de Submódulo utilizamos la opción `--recurse-submodules` para comprobar que todo nuestro trabajo en submódulos se ha publicado antes de subir el superproyecto, lo cual puede ser muy útil cuando se utilizan submódulos.

En Otros puntos de enganche del lado cliente hablamos brevemente sobre el gancho `pre-push`, que es un script que podemos establecer para que se ejecute antes de que una subida se complete para verificar qué se debe permitir subir.

Por último, en Enviando (push) referencias echamos un vistazo a la subida con un refspec completo en lugar de los atajos generales que se utilizan normalmente. Esto puede ayudar a ser muy específico acerca de qué trabajo se desea compartir.

git remote

El comando `git remote` es una herramienta de gestión para el registro de repositorios remotos. Esto te permite guardar largas direcciones URL como cortos manejadores (handles), tales como '*origin*', para que no tengas que escribir las URL todo el tiempo. Puedes tener varios de estos y el comando `git remote` se utiliza para añadir, modificar y borrarlos.

Este comando se trata en detalle en Trabajar con Remotos, incluyendo listar, añadir, eliminar y cambiar el nombre de ellos.

Se utiliza en casi todos los capítulos subsiguientes en el libro también, pero siempre en la formato estándar `git remote add <nombre> <url>`.

git archive

El comando `git archive` se utiliza para crear un archivo empaquetado de una instantánea específica del proyecto.

Usamos `git archive` para crear un tarball de un proyecto para su compartición en Preparando una versión.

git submodule

El comando `git submodule` se utiliza para gestionar repositorios externos dentro de repositorios normales. Esto podría ser por bibliotecas u otros tipos de recursos compartidos. El comando `submodule` tiene varios sub-commandos (`add`, `update`, `sync`, etc) para la gestión de estos recursos.

Este comando es sólo mencionado y cubierto enteramente en Submódulos.

Inspección y Comparación

git show

El comando `git show` puede mostrar un objeto Git de una manera simple y legible por humanos. Normalmente se usaría esto para mostrar la información sobre una etiqueta o un commit.

Primero lo usamos para mostrar información de etiqueta con anotaciones en Etiquetas Anotadas.

Más tarde lo usamos bastante en Revisión por selección para mostrar las confirmaciones que nuestras diversas selecciones de revisión resuelven.

Una de las cosas más interesantes que ver con `git show` está en Re-fusión Manual de Archivos para extraer contenidos de un archivo específico de diversas etapas durante un conflicto de fusión.

git shortlog

El comando `git shortlog` se utiliza para resumir la salida de `git log`. Toma muchas de las mismas opciones que el comando `git log` pero, en lugar de enumerar todos las commits, presentará un resumen de los commits agrupados por autor.

Mostramos cómo usarlo para crear un buen registro de cambios en El registro resumido.

git describe

El comando `git describe` se utiliza para tomar cualquier cosa que remite a un commit y producir una cadena de texto que es de alguna manera legible por humanos y no va a cambiar. Es una manera de obtener una descripción de un commit que es tan inequívoca como el SHA-1 del commit, pero más comprensible.

Utilizamos `git describe` en Generando un número de compilación y Preparando una versión para obtener una cadena de texto para nombrar nuestro archivo de liberación.

Depuración

Git tiene un par de comandos que se utilizan para ayudar a depurar un problema en tu código. Esto va desde averiguar donde algo se introdujo a averiguar quién lo introdujo.

git bisect

La herramienta `git bisect` es una herramienta de depuración increíblemente útil, utilizada para

encontrar qué commit específico fue el primero en introducir un bug o problema, haciendo una búsqueda binaria automática.

Está completamente cubierto de Búsqueda binaria y sólo se menciona en esa sección.

git blame

El comando `git blame` toma nota de las líneas de cualquier archivo con cual commit fue el último en introducir un cambio en cada línea del archivo y qué persona fue autor de ese commit. Esto es muy útil con el fin de encontrar a la persona para pedir más información sobre una sección específica de tu código.

Se cubre en Anotaciones de archivo y sólo se menciona en esa sección.

git grep

El comando `git grep` puede ayudarte a encontrar cualquier cadena o expresión regular en cualquiera de los archivos en tu código fuente, incluyendo versiones más antiguas de tu proyecto.

Está cubierto en Git Grep y sólo se menciona en esa sección.

Parcheo

Unos comandos de Git se centran en el concepto de interpretar los commits en términos de los cambios que introducen, concibiendo las series de commit como series de parches. Estos comandos te ayudan a administrar tus ramas de esta manera.

git cherry-pick

El comando `git cherry-pick` se utiliza para tomar el cambio introducido en un único commit de Git y tratar de volver a introducirlo como un nuevo commit en la rama donde estás actualmente. Esto puede ser útil para escoger solamente uno o dos commits de una rama individual en lugar de fusionar la rama que contiene todos los cambios.

Esto se describe y se muestra en Flujos de trabajo reorganizando o entresacando.

git rebase

El comando `git rebase` es básicamente un `cherry-pick` automatizado. Determina una serie de commits y luego los escoge uno a uno en el mismo orden en otro lugar.

El rebasing se cubre en detalle en Reorganizar el Trabajo Realizado, inclusive cubriendo las incidencias de colaboración involucradas con el rebasing de ramas que ya son públicas.

Lo usamos en la práctica durante un ejemplo de dividir la historia en dos repositorios separados en Replace, utilizando el indicador `--onto` también.

Experimentamos la ejecución de un conflicto de fusión durante el rebasing en Rerere.

También lo usamos en un modo de secuencias de comandos interactiva con la opción `-i` en

Cambiando la confirmación de múltiples mensajes.

git revert

El comando `git revert` es esencialmente un `git cherry-pick` inverso. Crea un nuevo commit que se aplica exactamente al contrario del cambio introducido en el commit que estás apuntando, esencialmente deshaciendo o revertiéndolo.

Utilizamos éste en Revertir el compromiso para deshacer un commit de fusión.

Correo Electrónico

Muchos proyectos Git, incluido el propio Git, se mantienen totalmente a través de listas de correo. Git tiene una serie de herramientas integradas en él que ayudan a hacer más fácil este proceso, desde la generación de parches que pueden enviarse fácilmente por email a aplicar esos parches desde una casilla de correo electrónico.

git apply

El comando `git apply` aplica un parche creado con `git diff` o incluso el comando diff de GNU. Es similar a lo que el comando `patch` podría hacer con algunas pequeñas diferencias.

Mostramos a usarlo y las circunstancias en las que puedes hacerlo en Aplicando parches recibidos por e-mail.

git am

El comando `git am` se utiliza para aplicar parches desde una bandeja de entrada de correo electrónico, en concreto una que tenga formato mbox. Esto es útil para recibir parches por correo electrónico y aplicarlos a tu proyecto fácilmente.

Cubrimos el uso y flujo de trabajo en torno a `git am` en Aplicando un parche con `am` incluyendo el uso de las opciones `--resolved`, `-i` y `-3`.

Hay también una serie de ganchos (hooks) que se pueden utilizar para ayudar con el flujo de trabajo en torno a `git am` y todos ellos se cubren en Puntos en el flujo de trabajo del correo electrónico.

También lo utilizamos para aplicar parcheado con formato de cambios de GitHub Pull Request en Notificaciones por correo electrónico.

git format-patch

El comando `git format-patch` se utiliza para generar una serie de parches en formato mbox que puedes utilizar para enviar a una lista de correo con el formato correcto.

Examinamos un ejemplo de contribución a un proyecto mediante el uso de la herramienta `git format-patch` en Proyecto público a través de correo electrónico.

git send-email

El comando `git send-email` se utiliza para enviar parches que son generados con `git format-patch` por correo electrónico.

Examinamos un ejemplo de contribución a un proyecto mediante el envío de parches con la herramienta `git send-email` en Proyecto público a través de correo electrónico.

git request-pull

El comando `git request-pull` se utiliza simplemente para generar un cuerpo de mensaje de ejemplo para enviar por correo electrónico a alguien. Si tienes una rama en un servidor público y quieres que alguien sepa cómo integrar esos cambios sin enviar los parches a través de correo electrónico, puedes ejecutar este comando y enviar el resultado a la persona que deseas que reviva (pull) los cambios.

Mostramos como usar `git request-pull` para generar un mensaje pull en Proyecto público bifurcado.

Sistemas Externos

Git viene con unos pocos comandos para integrarse con otros sistemas de control de versiones.

git svn

El comando `git svn` se utiliza para comunicarnos como cliente con el sistema de control de versiones Subversion.

Esto significa que puedes usar Git para obtener desde y enviar a un servidor Subversion.

Este comando es cubierto en profundidad en Git y Subversión.

git fast-import

Para otros sistemas de control de versiones o importación desde prácticamente cualquier formato, puedes usar `git fast-import` para convertir rápidamente el otro formato a algo que Git pueda registrar fácilmente.

Este comando es cubierto en profundidad en Un proveedor individual.

Administración

Si estás administrando un repositorio Git o necesitas arreglar algo a lo grande, Git ofrece una serie de comandos de administración para asistirte.

git gc

El comando `git gc` ejecuta la *'recogida de basura'* en tu repositorio, eliminando los archivos innecesarios en tu base de datos y empaquetando los archivos restantes en un formato más

eficiente.

Este comando se ejecuta normalmente en segundo plano, aunque se puede ejecutar manualmente si se desea. Repasamos algunos ejemplos de esto en Mantenimiento.

git fsck

El comando `git fsck` se utiliza para comprobar la base de datos interna en busca de problemas o inconsistencias.

Sólo lo utilizamos una vez de forma rápida en Recuperación de datos para buscar objetos colgantes.

git reflog

El comando `git reflog` examina un registro de donde han estado todas las cabezas de tus ramas mientras trabajas para encontrar commits que puedes haber perdido a través de la reescritura de historias.

Cubrimos este comando principalmente en Nombres cortos de RefLog, donde mostramos el uso normal y cómo usar `git log -g` para ver la misma información con salida de `git log`.

También repasamos un ejemplo práctico de la recuperación de tal rama perdida en Recuperación de datos.

git filter-branch

El comando `git filter-branch` se utiliza para reescribir un montón de commits de acuerdo a ciertos patrones, como la eliminación de un archivo de todas partes o el filtrado de todo el repositorio a un solo subdirectorio para sacar un proyecto.

En Remover un archivo de cada confirmación explicamos el comando y exploramos varias opciones diferentes, tales como `--commit-filter`, `--subdirectory-filter` y `--tree-filter`.

En Git-p4 y TFS lo usamos para arreglar repositorios externos importados.

Comandos de Fontanería

También había un buen número de comandos de fontanería de bajo nivel que nos encontramos en el libro.

El primero que encontramos es `ls-remote` en Referencias de Pull Request, que usamos para mirar las referencias en bruto en el servidor.

Usamos `ls-files` en Re-fusión Manual de Archivos, Rerere y El Índice para echar un vistazo más en bruto a la apariencia del área de preparación.

También mencionamos `rev-parse` en Referencias por rama para tomar casi cualquier cadena y convertirla en un objeto SHA1.

Sin embargo, la mayoría de los comandos de bajo nivel de fontanería que cubrimos están en Los

entresijos internos de Git, que es más o menos en lo que el capítulo se centra. Tratamos de evitar el uso de ellos en la mayor parte del resto del libro.

Unresolved directive in progit_v2.1.1.asc - include::index.asc[]